Data Structures and Amortized Complexity in a Functional Setting

Part I: Theoretical Aspects

Berry Schoenmakers

Doctoral Dissertation from
Eindhoven University of Technology[1]

September 1992

[1]Current address: CWI, Kruislaan 413, 1098 SJ Amsterdam, Netherlands. E-mail: berry@cwi.nl

Chapter 1

Algebras

The diversity of studies of data structures, abstract data types, many-sorted algebras, etc. provides evidence for the fundamental role of such structures in computing science. Indeed, many designs of sophisticated algorithms hinge on the right choice of data structures and efficient implementation thereof. Such structures, consisting of a number of data types and operations, are the subject of this monograph; we call them *algebras* to reflect our mathematical view of these structures.

In descriptions of programming languages one generally opposes the data structures of the language to its control structures. This natural division between "data" and "control" can be traced back to primitive models of computation such as Turing machines. Taking Church's thesis for granted, Turing machines are powerful enough to describe any algorithm; hence, any algorithm can be considered as a composition from a relatively small repertoire of operations that manipulate data. A Turing machine consists of a (finite) control part and an (infinite) tape used to store data. Combined with a tape head, the tape forms a simple data structure with operations "read", "write", "move left" and "move right". By means of these operations the initial contents of the tape are transformed into the corresponding output according to the program in the control part. The distinction between control structures and data structures can also be recognized in the Von Neumann style of computers in use nowadays. Abstract models of this type of computers are known as "random access machines" (see e.g. [23]); the primitive data structure used in this model is an infinite array of cells, called memory, whose contents can be modified by means of move instructions and arithmetic instructions.

So, "data" and "control" can be viewed as two more or less orthogonal aspects, and therefore they are often studied independently. For instance, in semantics of imperative programs, the types of variables and the forms of expressions are usually ignored, but one concentrates on iteration and recursion constructs, say. In this thesis, we concentrate on data structures and to us it is important that a programming language provides—among other things—a number of algebras, which are abstractions of the more primitive algebras provided by machine languages.

The remainder of this chapter consists of two sections. In the next section we introduce some frequently used data types and operations. These types and operations are used in two ways: simply as mathematical objects, and as components of algebras or, more generally, as parts of algorithms. In the other section we introduce our notion of algebras and we provide some examples of typical algebras which can be implemented in many programming languages. Furthermore, a restricted form of algebras, called *monoalgebras*, is introduced for which some theory is developed in later chapters.

1.1 Basic data types and operations

In addition to standard notations for well-known mathematical objects such as sets and arithmetical operators, we introduce some home-grown notations, in particular for structured types like finite lists. We would like to stress that only notation matters to us in this section, not foundations; the concepts we use have been founded rigorously elsewhere, and we take that as a starting-point.

1.1.1 Tuples

We consider *tuples* as elements of cartesian products. The *empty tuple*, the only member of the empty cartesian product, is denoted by $\perp$. Nonempty tuples are either enclosed by a pair of parentheses or by a pair of angular brackets. To select the respective components of a nonempty tuple t we write $t.0$ to denote the first component, $t.1$ to denote the second one, and so on. For example, $\langle x, y\rangle.1 = y$.

1.1.2 Relations and functions

A *(binary) relation* is a set of ordered pairs. For relation R, we denote its *domain* $\{x, y : (x, y) \in R : x\}$ by $\operatorname{dom} R$, and its *range* $\{x, y : (x, y) \in R : y\}$ by $\operatorname{rng} R$. For $x \in \operatorname{dom} R$, we use $R.x$ (in addition to $R(x)$ and $R\ x$) to denote the *application of* R *to* x, which means that $R.x$ denotes *a* solution of the equation $y : (x, y) {\in} R$. In accordance with this notation for relation application—in which the "input" stands to the right of the relation's name—we introduce yRx as a shorthand for $(x, y){\in}R$. This notation goes well with the conventional notation for relation composition in which the rightmost relation is applied first:

$z\,(S \circ R)\,x \;\equiv\; (\exists\, y :: zSy \;\wedge\; yRx)$	*composition* $S \circ R$ of R and S
$y\,R^\star\,x \;\equiv\; xRy$	*dual* $R^\star$ of R
$y\,R{\restriction}X\,x \;\equiv\; yRx \;\wedge\; x \in X$	*restriction* $R{\restriction}X$ of R to X.

A relation R satisfying $yRx \Rightarrow y{=}x$ for all x and y, is called an *identity (relation)*. Hence, $=$ is an identity itself. A relation whose domain is a singleton set is said to have *no inputs*.

A relation f is called a *function* when it satisfies *Leibniz's rule*, which states that $x{=}y \Rightarrow f.x{=}f.y$ for all $x, y \in \operatorname{dom} f$. In other words, for a function f, $y f x$ equivales $x{\in}\operatorname{dom} f \wedge y{=}f.x$. With "relation" replaced by "function", the above introduced nomenclature for relations is also used for functions. A function without inputs is also called a *constant (function)*. Note that identities are functions. By $f[x{:=}y]$, with $x \in \operatorname{dom} f$, we denote the function equal to f except that x's image is y.

For sets X and Y, we distinguish the following kinds of relations and functions:

$$\begin{array}{lll}
\mathcal{P}(X{\times}Y) = \{R \mid \operatorname{dom} R \subseteq X \ \wedge\ \operatorname{rng} R \subseteq Y\} & & \text{relations on } X \text{ and } Y \\
X{\curvearrowright}Y = \{f \mid \operatorname{dom} f \subseteq X \ \wedge\ \operatorname{rng} f \subseteq Y\} & & \text{partial functions from } X \text{ to } Y \\
X{\to}Y = \{f \mid \operatorname{dom} f = X \ \wedge\ \operatorname{rng} f \subseteq Y\} & & \text{(total) functions from } X \text{ to } Y.
\end{array}$$

Note that $X{\to}Y \subseteq X{\curvearrowright}Y \subseteq \mathcal{P}(X{\times}Y)$, and also that $\emptyset$ is a (total) function of type $\emptyset{\to}Y$, a partial function of type $X{\curvearrowright}Y$, and a relation of type $\mathcal{P}(X{\times}Y)$, for every X and Y.

Remark 1.1
Apart from the different meaning of yRx we have adhered to conventional notations for relations and functions as much as possible. By using yRx as shorthand for $(x, y) \in R$, our notation is such that data flows from right to left on the *entire* level of application, while it flows in the opposite direction on the typing level; for instance, we write $z = g.(f.x)$ for functions $f \in X \to Y$ and $g \in Y \to Z$.
$\square$

1.1.3 Simple data types

To begin with, we have the *unit type* $\{\bot\}$. This trivial type serves as the domain of relations without inputs. These relations are of the form $\{\bot\} \times T$ with T nonempty, and we denote them by $?_T$ (subscript $_T$ is omitted when the type is clear from the context). We also use $?_T$ as abbreviation for $?_T.\bot$; in that case $?_T$ denotes an arbitrary value of type T. This value is not fixed, so we do not know whether $?_T = ?_T$—unless T is a singleton type, of course. Note that $?_T$ is a constant if T is a singleton type; in that case we write—as usual—c instead of $?_T$, where c is the unique element of T.

Another finite type is the set of *booleans* $\{\mathrm{false}, \mathrm{true}\}$, denoted by Bool. Boolean operators are denoted by the same symbols used for predicates, which are after all boolean-valued functions.

Finally, there are the *numerical types* Nat (naturals), Int (integers), and Real (reals), together with the well-known arithmetical operators. For these types we use standard notations, except perhaps for our notation for *intervals*: $[a..b)$ denotes set $\{x \mid a \leq x < b\}$, in which the type of dummy x depends on the context; the three other variations are $[a..b]$, $(a..b]$, and $(a..b)$, which speak for themselves.

In connection with numerical types we use the values ∞ and $-\infty$. As is common practice, it depends on the context whether a numerical type contains these values or not. For example, $+$ and $\max$ are both arithmetic operators of type $\mathrm{Int} \times \mathrm{Int} \rightarrow \mathrm{Int}$, but usually Int is equal to $(-\infty..\infty)$ in case of $+$, while it equals $[-\infty..\infty]$ for $\max$.

1.1.4 Structured data types

Let T be a nonempty type. In order of increasing structure, we have the finite sets over T, the finite bags over T, the finite lists over T, and the finite binary trees over T, or "sets", "bags", "lists", and "trees" for short. These so-called structured types are denoted by $\{T\}$, $\Lbag T \Rbag$, $[T]$, and $\langle T\rangle$, respectively, and in the sequel we will sometimes call elements of these types "structures". In accordance with this notation, we will also use the above pairs of brackets to form structures of the respective types. In particular, we use $\{\,\}$, $\Lbag\,\Rbag$, $[\,]$, and $\langle\,\rangle$ to denote the empty set, the empty bag, the empty list, and the empty tree. Similarly, $\{a\}$, $\Lbag a\Rbag$, $[a]$, and $\langle a\rangle$ denote singleton structures, and, for instance, $\Lbag 1,2,1\Rbag$ denotes a three-element bag. In this notation, expressions like $[\{0\}]$ denote both a type and a structure. The role of types and structures is however quite different such that the context usually resolves such ambiguities.

To denote the *size* of a structure, we use the symbol $\#$; in case of lists we also speak of "length" instead of "size". Furthermore, we write $(a\in)$ for the function that tells us whether a *occurs* in a structure. In case T is linearly ordered, we use $\downarrow x$ to denote the *minimum* value occurring in a nonempty structure x, and $\uparrow x$ to denote its *maximum* value. For some T, such as Int, minimum and maximum of the empty structure are defined as well, respectively equal to ∞ and $-\infty$.

So much for the common notations for the structured types. Next we introduce some notations which are specific to each of these types.

For finite sets we use, in addition to standard set notation, $\Downarrow S$ to denote set $S \setminus \{\downarrow S\}$, for nonempty S. As a restricted version of $\cup$ we have $\uplus$ which denotes *disjoint set union*; that is, $S \uplus T$ is defined if and only if $S \cap T = \{\,\}$. Hence, $\uplus$ is a partial function from $\{T\} \times \{T\}$ to $\{T\}$. Similarly, $\oplus$ and $\ominus$ with $S \oplus a = S \cup \{a\}$ and $S \ominus a = S \setminus \{a\}$ are partial set operations which are defined only if $a \notin S$ and $a \in S$, respectively.

For a bag B, $\Downarrow B$ stands for the bag obtained by removing a single occurrence of $\downarrow B$ from B: $\Downarrow B = B \ominus \Lbag\downarrow B\Rbag$, for B nonempty. Here, $\ominus$ denotes *bag subtraction*, and *bag summation* is denoted by $\oplus$.

For lists we have a quite comprehensive set of operations. First of all, to construct nonempty lists we have $\vdash$ (*cons*) and $\dashv$ (*snoc*): $a \vdash s$ denotes the list with *head* a and *tail* s, and $s \dashv a$ denotes the list with *front* s and *last element* a. Note that $[a]$ abbreviates both $a \vdash [\,]$ and $[\,] \dashv a$. Furthermore, $s \mathbin{+\!\!+} t$ denotes the *catenation* of lists s and t. To dissect nonempty lists, we use $hd.s$, $tl.s$, $ft.s$, $lt.s$, $s{\uparrow}n$, and $s{\downarrow}n$ $(0 \leq n \leq \#s)$ to select, respectively, the head, tail, front, last element, *prefix* of length n, and *suffix* of length $\#s-n$ of s. (Hence, $s = s{\uparrow}n \mathbin{+\!\!+} s{\downarrow}n$.) Also, we let $s.n$ stand for the $(n+1)$-st element of s $(0 \leq n < \#s)$, and $rev.s$ denotes the *reverse* of s.

Finally, for binary trees, we use $\langle t, a, u \rangle$ to denote a nonempty binary tree with *left subtree* t, *root* a, and *right subtree* u. To select the respective components of a nonempty tree, we use functions l, m, and r ("left", "middle", and "right"). Note that $\langle a \rangle$ is short for $\langle \langle\,\rangle, a, \langle\,\rangle \rangle$. In addition to $\#$, which denotes the actual size of a tree, it is often convenient to use the actual size *plus one* in efficiency analyses because it is positive. We denote it by $\sharp$, and it may be defined recursively by $\sharp\langle\,\rangle = 1$ and $\sharp\langle t, a, u \rangle = \sharp t + \sharp u$.

The common use of $\#$, $\in$, $\downarrow$, and $\uparrow$ for the structured types is justified by the following natural correspondence between these types. First of all, the *inorder traversal* of a tree converts a tree into a list:

$$\begin{array}{lcl} \overline{\langle\,\rangle} & = & [\,] \\ \overline{\langle t, a, u \rangle} & = & \overline{t} \mathbin{+\!\!+} [a] \mathbin{+\!\!+} \overline{u}. \end{array}$$

Similarly, a list can be converted into a bag:

$$\begin{array}{lcl} \overline{[\,]} & = & \Lbag\,\Rbag \\ \overline{a \vdash s} & = & \Lbag a \Rbag \oplus \overline{s}. \end{array}$$

And, finally, a bag can be converted into a set:

$$\begin{array}{lcl} \overline{\Lbag\,\Rbag} & = & \{\,\} \\ \overline{\Lbag a \Rbag \oplus B} & = & \{a\} \cup \overline{B}. \end{array}$$

These conversions leave the set of values present in a structure intact, but each application of $\overline{\cdot}$ destroys some structure until that set is obtained. Note that $\overline{\cdot}$ also leaves the size of a structure intact, except that for bags $\#B$ is larger than $\#\overline{B}$ when B contains duplicates.

1.2 Algebras

Independent of a particular programming style, the following notion of algebras combined with a notion of algebra refinement is believed to be of great importance to algorithm design, since it provides a basis for a good separation of concerns.

Definition 1.2
An *algebra* A consists of a sequence of sets τ and a sequence of relations o. It is written

$$\mathsf{A} = (\,\tau \mid o\,).$$

The sets are called A's *data types*, and the relations are called A's *operations*.
□

For the scope of this chapter, the order of the data types and operations is irrelevant; it plays a role in our definition of algebra refinement in the next chapter. By removing data types and/or operations from an algebra and possibly reordering the data types and operations, we obtain, what we call, a *subalgebra*. Algebra $(\ |\ )$ is a subalgebra of every algebra.

The following examples give an idea of the aspects of algebras that are considered throughout this thesis.

Example 1.3
One of the most basic algebras is the algebra of booleans:

$$(\,\mathrm{Bool} \mid \mathrm{false}, \neg, \wedge\,).$$

Other boolean operations like $\vee$ and $\equiv$ can be expressed in terms of the operations of this algebra. For example, true may be obtained as $\neg$false, and $a \vee b$ is equivalent to $\neg(\neg a \wedge \neg b)$. So, the operations of this algebra form a basis for the boolean operations.

It is in general a good idea to keep the set of operations of an algebra small and simple but, unfortunately, this advice does not help in all circumstances. For instance, an equally powerful algebra, using the well-known "nand" operator (the composite of $\wedge$ and $\neg$), is:

$$(\,\mathrm{Bool} \mid \mathrm{false}, \neg \circ \wedge\,).$$

Although this algebra has only two operations, it is more complicated to express the other boolean operations in terms of these two operations.
□

In the algebra of booleans, Bool is the only data type that matters[1], but in general the operations involve more than one data type. However, when implementing such a set of operations, we concentrate on a single data type—common to all operations—that has to be implemented. This data type then becomes the data type of the algebra, and the implementation of the remaining types is taken for granted.

Example 1.4
Algebras involving numerical types are available in almost any programming language. Here is a very simple one:

$$(\,\mathrm{Nat} \mid 0, (+1), (-1), (=0)\,).$$

We remark that (-1) is a partial function w.r.t. the data type of this algebra; its domain is $\mathrm{Nat}\backslash\{0\}$.

Numbers are usually represented by digit sequences. A *unary* implementation of the above algebra is for example:

[1] Actually, this is not quite true. See Remark 2.24

$$(\,[\{0\}] \mid [\,], (\dashv 0), ft, (=[\,])\,).$$

In this implementation, natural number n is represented by $(\dashv 0)^n.[\,]$, the list of n zeros. A *binary* implementation might look as follows:

$$(\,[\{0,1\}] \mid \mathsf{zero}, \mathsf{suc}, \mathsf{pred}, \mathsf{iszero}\,),$$

with

$$\begin{array}{lcl}
\mathsf{zero} & = & [\,] \\
\mathsf{suc}.[\,] & = & [1] \\
\mathsf{suc}.(s \dashv 0) & = & s \dashv 1 \quad , s \neq [\,] \\
\mathsf{suc}.(s \dashv 1) & = & \mathsf{suc}.s \dashv 0 \\
\mathsf{pred}.[1] & = & [\,] \\
\mathsf{pred}.(s \dashv 1) & = & s \dashv 0 \quad , s \neq [\,] \\
\mathsf{pred}.(s \dashv 0) & = & \mathsf{pred}.s \dashv 1 \\
\mathsf{iszero}.s & = & s = [\,].
\end{array}$$

Only binary lists containing at least one 1 are in the domain of pred (hence, $[\,] \notin \mathrm{dom}\,\mathsf{pred}$). The non-recursive alternatives in the definition of pred will therefore terminate any evaluation of pred.

A more useful numerical algebra is for instance:

$$(\,\mathrm{Int} \mid 1, +, -, *, \mathrm{div}\,, \mathrm{mod}\,, \mathrm{max}, \mathrm{min}, =, <, >\,).$$

A drawback of this algebra is that each integer has to be generated from the integer 1; e.g., 0 may be obtained as $1-1$. In many programming languages numerical values can be generated from their decimal representation. This facility may be provided as follows:

$$(\,\mathrm{Int} \mid \mathit{Dec2Int}, \mathit{Int2Dec}, +, -, *, \mathrm{div}\,, \mathrm{mod}\,, \mathrm{max}, \mathrm{min}, =\,).$$

Operations *Dec2Int* and *Int2Dec* convert types $[[0..10)]$ and Int into each other.

For reals we also have such conversion operations. In most languages only a subset of the reals can be generated in this way, and operations return approximations of the exact result. Some languages, however, offer exact real-arithmetic. For instance, the following algebra can be available:

$$(\,\mathrm{Real} \mid e, \pi, nth\,),$$

where $nth \in \mathrm{Nat}\times\mathrm{Real} \to [0..10)$ and $nth.n.\alpha$=“$(n{+}1)$-st digit of the fractional part of the decimal representation of α”.

To describe conversions between integers and reals, say, we need an algebra with more than one data type. Given algebras $(\,\mathrm{Int} \mid \sigma\,)$ and $(\,\mathrm{Real} \mid \varsigma\,)$, the following algebra is appropriate:

$$(\,\mathrm{Int}, \mathrm{Real} \mid \sigma, \varsigma, \mathit{Int2Real}, \mathit{Real2Int}\,).$$

The conversion from Real to Int is for instance done by rounding, and the reverse conversion may simply be the identity on Int.
□

Many of the aspects touched upon in the previous example will be ignored in the rest of this thesis. That is, the ins and outs of built-in algebras will not be discussed much further, but we will mainly concentrate on efficient implementations of more abstract algebras.

Example 1.5
The algebra of *stacks* is usually provided as "kernel" for the list operations in functional languages. A possible definition is

$$([T] \mid [\,], (=[\,]), \vdash, hd, tl),$$

for arbitrary type T. The operations of this algebra are usually assumed to have $O(1)$ time complexity. Other list algebras like the algebra of *queues*, defined by

$$([T] \mid [\,], (=[\,]), \dashv, hd, tl),$$

can be implemented using this kernel. Since $\dashv$ may be expressed in terms of the stack-operations and $\vdash$ may be expressed in terms of the queue-operations, both algebras are equally powerful. Taking efficiency into consideration, however, gives a different picture. Programming $\dashv$ in terms of stack-operations will lead to a linear time program, while the algebra of queues can be implemented such that $\dashv$ takes constant time only (see Section 4.5).

Examples of more advanced list algebras that we shall encounter in the sequel are concatenable queues (in Section 4.5):

$$([T] \mid [\,], (=[\,]), \dashv, hd, tl, +\!\!+),$$

and stacks extended with the minimum operation (in Chapter 8):

$$([T] \mid [\,], (=[\,]), \vdash, hd, tl, \downarrow),$$

where T is assumed to be linearly ordered so that $\downarrow$ is defined.
□

The above list algebras are all equally powerful in the sense that their operations are rich enough to program any operation on lists. The relative efficiencies of (the implementations of) these algebras, however, make up the difference. For example, operation $\downarrow$ is included as additional operation in the last algebra because it is possible that there exist more efficient ways of implementing $\downarrow$ than by just programming it in terms of the stack-operations. In general, an operation is not only included in an algebra because it cannot be programmed in terms of the other operations, but also because it cannot be programmed *without loss of efficiency* in terms of the other operations!

Algebras involving sets and bags are usually not provided by programming languages, but they are used frequently in (abstract) programs.

Example 1.6
The following algebra is one of the many variations of so-called *priority queues*:

$$(\,\Lbag \mathrm{Int} \Rbag \mid \Lbag\,\Rbag, \Lbag\cdot\Rbag, \oplus, (=\Lbag\,\Rbag), \downarrow, \Downarrow).$$

Implementations of priority queues often use some kind of trees to represent bags. In Part II we will give several examples thereof.
□

Arrays and pointers are two important data types provided by most imperative languages. As algebras they might look as follows.

Example 1.7
For natural N and nonempty type T, we consider an algebra with data type $[0..N) \to T$. Elements of this data type are called *arrays*. For array a we use $a[i]$ as alias for $a.i$, $0 \le i < N$, and $a[i{:=}x]$ denotes the array equal to a except that $a[i{:=}x][i] = x$. Furthermore, there is operation ? (cf. Section 1.1.3) to create an arbitrary array which may serve as an initial value. In summary, the algebra of arrays looks like

$$(\,[0..N) \to T \mid ?, \mathit{lookup}, \mathit{update}\,),$$

with $\mathit{lookup}.a.i = a[i]$ and $\mathit{update}.a.i.x = a[i{:=}x]$. Usually all operations of this algebra are assumed to have $O(1)$ time complexity. Therefore, ? is a relation since if it were to be a function, evaluation of ? should always return the same value (of type $[0..N) \to T$), but this requires $O(N)$ time for the standard implementation of arrays. So, this is an example of an algebra with a relational operation: the outcome of $?[0] = ?[0]$, for example, is indeterminate. Furthermore, evaluation of $a[i{:=}x]$ will in general destroy the representation of a, since the value of $a[i]$ is simply overwritten to achieve $O(1)$ time complexity. The usage of such *destructive* operations will be discussed in Chapter 4.
□

Example 1.8
Pointer structures are similar to arrays in the sense that they can be viewed as functions too. The difference is that arrays are total functions on a finite interval of integers, whereas pointer structures are partial functions on some, usually anonymous, domain. To sketch the idea behind pointer types we consider the following algebra:

$$(\,\Omega, \Omega \frown T \mid \mathrm{nil}, =, \{(\mathrm{nil}, ?_T)\}, \mathit{value}, \mathit{new}, \mathit{assign}, \mathit{dispose}\,),$$

where

Ω is some set ("addresses of memory cells");
T is some nonempty type ("contents of memory cells");
$\mathrm{nil} \in \Omega$;

$=$ denotes equality on Ω;
$value \in \Omega\times(\Omega\curvearrowright T)\curvearrowright T$ is defined by $value.p.s = s.p$, for $p \in \mathrm{dom}\, s$;
$new \subseteq (\Omega\curvearrowright T) \times (\Omega\times(\Omega\curvearrowright T))$ is defined by $new.s = (p, s')$ with $p \notin \mathrm{dom}\, s$ and $s' = s \cup \{(p, ?_T)\}$, for $\mathrm{dom}\, s \neq \Omega$;
$assign \in \Omega\times T\times(\Omega\curvearrowright T)\curvearrowright(\Omega\curvearrowright T)$ is defined by $assign.p.x.s = s[p{:=}x]$, for $p \in \mathrm{dom}\, s$;
$dispose \in \Omega\times(\Omega\curvearrowright T)\curvearrowright(\Omega\curvearrowright T)$ is defined by $dispose.p.s = s\backslash\{(p, s.p)\}$, for $p \in \mathrm{dom}\, s \backslash\{\mathrm{nil}\}$.

Elements of Ω are *pointers* and an element of $\Omega\curvearrowright T$ can be considered as a piece of *memory.* Operation *new* is not a function: the outcome of $new.s$ not only depends on the value of s but also on the way this value has been obtained.

Type T often refers to Ω. For example, $T = \mathrm{Int}\times\Omega$ for singly-linked lists, and in such a case several linked lists may be represented in the same part of memory; think, for instance, of an array of linked-lists. The advantage of a pointer type (over an array type) is that the same part of memory, namely Ω, can be used to represent a number of lists with a total number of $\#\Omega$ elements, where the distribution over the lists evolves dynamically, depending on the input.

Note that $\emptyset \in \Omega\curvearrowright T$ cannot be constructed by means of the operations of this algebra; in particular, nil is always in the domain. Note that we allow *value* and *assign* to operate on nil.
□

In Section 4.3, we will introduce some Pascal-like notation for pointers in an informal way, and on the whole we will not be very formal about pointers.

1.3 Monoalgebras

Instead of the general notion of algebras as defined in Definition 1.2, we will use the following restricted version as our "working definition" throughout Part I.

Definition 1.9
A *monoalgebra* A is an algebra with a *single, nonempty* data type A and a *finite* number of operations. The operations should be *functional*, and *first-order with respect to A*. An operation is called first-order w.r.t. A when both its domain and its range are (subsets of) cartesian products composed of data type A and data types of other algebras not involving algebra A. As representatives of such operations we take

(a) *creations* of type $T \to A$

(b) *transformations* of type $A\times A\curvearrowright A$

(c) *inspections* of type $A \to T$,

where T stands for data types of algebras not involving A.
□

Remark 1.10
As is common practice in definitions of programming languages, algebras are identified by the names of their data types, since the data type uniquely determines the set of operations belonging to it. *In general, however, we should use algebras instead of their data types to type operations.* In Definition 1.9, for instance, A and T may be equal when viewed as sets, but they should be data types of different algebras—or else creations and inspections are indistinguishable.
□

Generally, creations and transformations are used to generate values of the data type, whereas inspection operations are used to discriminate values. Characteristic of inspections is that A does not occur in the range. The difference between creations and transformations is that A does not occur in the domain of a creation, whereas it occurs at least once in the domain of a transformation. The representatives in Definition 1.9 have been chosen such that the definitions and proofs involving monoalgebras (particularly, in Chapter 2) can be limited to these cases. The requirement that operations of monoalgebras should be first-order w.r.t. the data type of the algebra means, informally, that the arguments of the operations should be "first-order data": the domains and ranges of first-order operations are cartesian products of A and other types, but a function type like $A \to A$ is not allowed as component of such a cartesian product.

However, operations of type $[A] \to A$, $\lfloor A \rfloor \to \langle A \rangle$, $\{\perp\} \to [\{A\}]$, etc., are also considered as first-order w.r.t. A: in all these cases, arguments involving type A are exclusively manipulated by the operation itself, not by other arguments of the operation. The following example deals with such an operation.

Example 1.11
A well-known higher-order operation on lists is $\star$ ("map"), defined by $(f \star s).i = f.(s.i)$ for $0 \leq i < \#s$. Such an operation may be added to the algebra of stacks:

$$(\,[T] \mid [\,], (=[\,]), \vdash, hd, tl, \star\,),$$

with $\star$ of type $(T{\to}T) \times [T] \to [T]$. Although $\star$ takes a function as parameter, implementation of the above algebra is not difficult because the implementation of the function parameter has to be taken care of elsewhere. That is, given a program for f, $(f\star)$ can be programmed in terms of the stack-operations. In our terminology, $\star$ is therefore *not* a higher-order operation w.r.t. type $[T]$. Still, operations like $\star$ will not receive much attention in the sequel.
□

In the previous section some algebras have been exhibited that violate the restrictions imposed in the definition of monoalgebras. In Example 1.7, the algebra of arrays provides an example of an algebra with a nonfunctional operation, though a very simple one. Operation *new* in Example 1.8 is also nonfunctional. At the end of Example 1.4 we have seen an algebra with more than one data type, and in the next example we encounter an algebra with a higher-order operation w.r.t. its data type. In pathological cases we may also encounter algebras with an empty data type (Example 2.18).

Besides the fact that most algebras in this thesis are monoalgebras, most of them also share the property that the elements of their data types are objects of finite size. Algebra $(\,\mathrm{Real} \mid e, \pi, nth\,)$ from Example 1.4 is an example of a monoalgebra with objects of infinite size. Another illustration of objects of infinite size is presented in the next example, but there the algebra is not a monoalgebra.

Example 1.12
Let L denote the set of *infinite* lists over some anonymous universe. To generate infinite lists we use a fixpoint constructor μ of type $(L \to L) \frown L$, which returns a solution of equation $s : s = F.s$, where $F \in L \to L$ is such that this equation has a unique solution (see e.g. [17, Section 5.5]). In some functional programming languages the following algebra is then available:

$$(\,L \mid \mu, hd, tl, \vdash\,).$$

Note that μ is a *higher-order* operation w.r.t. L, since it takes a function on L as parameter; therefore, this is not a monoalgebra.

An example of an infinite list created by means of μ is $\mu.(1\vdash)$, the infinite lists of ones. Another example is the sorted list of naturals, corresponding to the function F given by $F.s = 0 \vdash (+1) \star s$.
□

In the next chapters we concentrate on monoalgebras. As will become apparent in the sequel, the essential point about a monoalgebra is that a number of connected operations involving the same data type are grouped together. Typical properties of monoalgebras are that all elements of its data type can be constructed by means of its operations, and also that distinct elements of the data type can be distinguished by means of (compositions of) the operations of the algebra. Removal or addition of a single operation may affect these properties drastically—as we shall see in the next chapter. In the design of data structures, it is also a well-known phenomenon that one or two extra operations may complicate efficient implementations of an algebra significantly.

Chapter 2

Algebra refinement

The main reason to introduce algebra refinement is that we want to use it as a formal, yet powerful and flexible, mechanism for specifying data structures. A specification of a data structure then asks for a *concrete* refinement of a given *abstract* algebra or, less formally, it asks for an implementation of a given algebra. An obvious characteristic of this approach to the design of data structures is that algebras play the role of specifications as well as the role of implementations.

As the majority of algebras in Part II of this thesis are monoalgebras, we shall define refinement only for this restricted class of algebras (Section 2.4). To prepare for this definition, we first discuss refinement of functions (and functional programs). In Section 2.3 we introduce data refinement as a generalization of algorithmic refinement, and—as stepping-stone towards our definition of algorithmic refinement in Section 2.2—specifications of functions are introduced in the next section.

2.1 Functional programs and specifications

We do not present a definition of a program notation in this chapter, because neither the actual borderline between abstract and concrete nor the efficiency of concrete programs is relevant for the development of a refinement theory—in contrast to the use of such a theory. What is more, the notion of a program is decoupled from the notion of a program notation in our refinement theory.

Definition 2.1
A *program* is a definition of a relation. If the defined relation is functional, then the program is called *functional* as well. The name of a program is the name of the relation defined by it. Relative to a particular program notation, a program is called *concrete* if it is expressed in the program notation; otherwise, it is called *abstract*.
□

In this chapter we deal with functional programs only; therefore we just use "program" instead of "functional program", or even "function", since in many contexts the actual definition of a function is irrelevant. Furthermore, we will use "abstract data types" and "concrete data types", "abstract algebras" and "concrete algebras", and so on, all relative to a particular program notation. We also use the terms "abstract" and "concrete" to refer to the refined object and its refinement, respectively; with respect to many program notations this is correct, but in some refinements this use of "abstract" and "concrete" does not comply with Definition 2.1. This is believed to cause no major difficulties.

As follows from its definition in Section 1.1.2, a function is completely defined by its domain and a function value for each value in the domain. In a specification we make a similar division:

Definition 2.2
A *specification* is a pair $\langle D, P\rangle$, in which D is a set and P is a predicate (on functions whose domain include D) satisfying

$$(\forall f, g : D \subseteq \operatorname{dom} f \;\wedge\; D \subseteq \operatorname{dom} g \;\wedge\; f{\restriction}D = g{\restriction}D : P(f) \equiv P(g)).$$

□

The restriction is imposed on $\langle D, P\rangle$ to exclude pathological specifications in which $P(f)$ depends on values $f.x$ where $x \notin D$. Usually, this restriction is trivially satisfied:

Property 2.3
For any set D and predicate Q (of the appropriate type),

$$\langle D, (\lambda f : D \subseteq \operatorname{dom} f \;: (\forall x : x \in D : Q(x, f.x)))\rangle$$

is a specification.
□

Specifications of this form may be called "pointwise" specifications because each function value is specified independent of the other function values. Note that not every specification can be written in this form. For instance, specification $\langle \mathrm{Bool}, (\lambda f : \mathrm{Bool} \subseteq \operatorname{dom} f \;: f.\mathrm{false} \neq f.\mathrm{true})\rangle$ cannot be written as a pointwise specification.

Definition 2.4
Function f is said to *satisfy* specification $\langle D, P\rangle$ when

$$D \subseteq \operatorname{dom} f \;\wedge\; P(f).$$

□

Instead of writing specifications as pairs, we usually formulate them in a less strict way. We write, for example: "Design a program for function $sum \in [\mathrm{Int}]{\rightarrow}\mathrm{Int}$ satisfying $(\forall s : s \in [\mathrm{Int}] : sum.s = (\Sigma i : 0 \leq i < \#s : s.i))$". In this specification sum is a dummy, so we are free to use any name we like for our solutions; in particular, we can use different names to distinguish successive refinements of the same specification. Moreover, we tacitly imply by this formulation that a function with a domain *larger* than [Int] is also satisfactory.

2.2 Algorithmic refinement of functions

Function g is called an algorithmic refinement of function f when every specification satisfied by f is satisfied by g as well. More formally:

Definition 2.5
Function f is said to be *algorithmically refined* by function g when

(1) $(\forall\, D, P : f \text{ satisfies } \langle D, P\rangle : g \text{ satisfies } \langle D, P\rangle)$.

□

Using this definition of algorithmic refinement, the verification of a particular refinement is rather cumbersome. Fortunately, we can derive, on account of the restriction in Definition 2.2, the following more useful characterizations of algorithmic refinement:

Property 2.6
Predicate (1) is equivalent to

(2) $\operatorname{dom} f \subseteq \operatorname{dom} g \;\wedge\; (\forall\, x : x \in \operatorname{dom} f : f.x = g.x)$,

and to

(3) $f \subseteq g$.

Proof To prove that (1) implies (2) for any f and g, we instantiate (1) with specification

$$\langle D, P\rangle = \langle \operatorname{dom} f, (\lambda\, h : \operatorname{dom} f \subseteq \operatorname{dom} h : (\forall\, x : x \in \operatorname{dom} f : f.x = h.x))\rangle .$$

Then f satisfies $\langle D, P\rangle$, hence so does g, which is equivalent to (2).

Next, assume (2) for some f and g. Then we observe for any specification $\langle D, P\rangle$:

$$\begin{array}{ll}
 & f \text{ satisfies } \langle D, P\rangle \\
\equiv & \{ \text{ Definition 2.4 } \} \\
 & D \subseteq \operatorname{dom} f \;\wedge\; P(f) \\
\Rightarrow & \{ \text{ (2), hence } D \subseteq \operatorname{dom} g \text{ and } f{\restriction}D = g{\restriction}D\text{; Definition 2.2 } \} \\
 & D \subseteq \operatorname{dom} g \;\wedge\; P(g) \\
\equiv & \{ \text{ Definition 2.4 } \} \\
 & g \text{ satisfies } \langle D, P\rangle ,
\end{array}$$

which settles the proof of the equivalence of (1) and (2).

The equivalence of (2) and (3) is obvious.

□

From (2) we infer that an algorithmic refinement is allowed to have a larger domain. Characterization (3) is more appropriate than (2) for deriving properties of algorithmic refinement. For instance, from (3) it follows immediately that algorithmic refinement induces a *partial order* on functions. Moreover, any application of f may be replaced by an application of g when $f \subseteq g$. In particular, composition is *monotonic* w.r.t. algorithmic refinement.

Property 2.7

(a)	$f \subseteq f$	reflexivity
(b)	$f \subseteq g \ \wedge \ g \subseteq f \ \Rightarrow \ f = g$	anti-symmetry
(c)	$f \subseteq g \ \wedge \ g \subseteq h \ \Rightarrow \ f \subseteq h$	transitivity
(d)	$f \subseteq g \ \Rightarrow \ f \circ h \subseteq g \circ h \ \wedge \ h \circ f \subseteq h \circ g$	monotonicity of $\circ$.

□

Characterization (2) is useful when one has to show that a particular *recursively* defined program is a correct refinement. For instance, the fact that function $g \in [\mathrm{Int}] \rightarrow \mathrm{Int}$, given by $g.[\,] = 0$ and $g.(a \vdash s) = a + g.s$, is a correct refinement of function f of the same type, defined by $f.s = (\Sigma\, i : 0 \leq i < \#s : s.i)$, is easily proved by establishing the second conjunct of (2) by induction on s.

The notion of algorithmic refinement enables us to specify a programming problem as a request for a concrete refinement of a given function. The summation problem from the previous section, for instance, may be specified by "Design a concrete refinement of program *sum* defined by $\mathrm{dom}\, sum = [\mathrm{Int}]$ and $sum.s = (\Sigma\, i : 0 \leq i < \#s : s.i)$". (Note that in this specification *sum* is *not* a dummy.) Since such specifications employ a notion of refinement of *functions*, they are bound to be deterministic. Since we regard such a specification method too restrictive, we have introduced nondeterministic specifications as well. In the next section we shall see that the notion of data refinement also enables a form of nondeterministic specifications that is especially suited for specifying data structures.

2.3 Data refinement of functions

The notion of algorithmic refinement enables us to relate the successive programs in a program development. The types of these programs are strongly related: the domains form an ascending sequence with respect to set inclusion. The types of the *components* of these programs may, however, be quite different. For example, suppose $f_1 \circ f_0 \in X \rightarrow Y$ is refined by $g_1 \circ g_0 \in X \rightarrow Y$, with $f_0 \in X \rightarrow A$, $f_1 \in A \rightarrow Y$, $g_0 \in X \rightarrow C$, and $g_1 \in C \rightarrow Y$. Then neither f_0 and g_0 nor f_1 and g_1 can be related when types A and C are incompatible. In such a refinement, A is typically the more abstract type and C the more concrete type. To relate such types and functions the notion of *data refinement* is introduced.

We introduce data refinement as a generalization of algorithmic refinement. For this purpose we rewrite characterization (2) of algorithmic refinement as follows:

$$(\forall\, x,y : x \in \operatorname{dom} f \;\;\wedge\;\; x = y : y \in \operatorname{dom} g \;\;\wedge\;\; f.x = g.y).$$

To relate functions of different types, we replace the equality signs by arbitrary relations (and rename the dummies for convenience later on):

Definition 2.8
Function f is said to be *data-refined* by function g under relations R and S, when

(4) $\quad (\forall\, a,c : a \in \operatorname{dom} f \;\;\wedge\;\; a\,R\,c : c \in \operatorname{dom} g \;\;\wedge\;\; f.a\,S\,g.c).$

□

In the above context, relations R and S are called coupling relations, or just *couplings*. When discussing (data) refinements we will often abuse the terms "abstract" and "concrete" to distinguish the refined objects from their refinements. The refined objects are called abstract and their refinements concrete, although this might give rise to "inconsistencies": e.g., in one refinement a data type is treated as abstract, while it is considered as concrete in another refinement.

Usually, couplings R and S are strongly related or even identical. Furthermore, identities are often used as couplings.

Example 2.9 (see Example 1.4)
In many applications of Definition 2.8, the couplings between the domains and ranges of the functions are described in terms of a relation between an abstract and a concrete type. In the specification below, for example, Nat is the abstract type and C the specified concrete type.

> Design a concrete type C, a relation $\simeq$ between Nat and C, and concrete functions $\mathsf{zero} \in C$, $\mathsf{iszero} \in C \curvearrowright \mathrm{Bool}$, $\mathsf{suc} \in C \curvearrowright C$, and $\mathsf{pred} \in C \curvearrowright C$ satisfying
>
> $$\begin{array}{l} 0 \simeq \mathsf{zero} \\ (\forall\, n,c : n \simeq c : c \in \operatorname{dom} \mathsf{iszero} \;\;\wedge\;\; n{=}0 \equiv \mathsf{iszero}.c) \\ (\forall\, n,c : n \simeq c : c \in \operatorname{dom} \mathsf{suc} \;\;\wedge\;\; n{+}1 \simeq \mathsf{suc}.c) \\ (\forall\, n,c : n{\neq}0 \;\;\wedge\;\; n \simeq c : c \in \operatorname{dom} \mathsf{pred} \;\;\wedge\;\; n{-}1 \simeq \mathsf{pred}.c). \end{array}$$

This specification contains four instances of (4). The identity on $\{\bot\}$ couples the domains of 0 and zero, which are, as usual, omitted. The identity on Bool serves as coupling between the ranges of $(=0)$ and iszero. Relation $\simeq$ couples the remaining domains and ranges.

A simple solution to this problem is to take a unary representation for naturals like type $[\{0\}]$. The coupling between Nat and $[\{0\}]$ is then defined by $n \simeq s \;\equiv\; n = \#s$. Under this coupling, we have the following refinements:

$0 \in \mathrm{Nat}$	$\mathsf{zero} \in [\{0\}]$	$\mathsf{zero} = [\,]$
$(=0) \in \mathrm{Nat}\to\mathrm{Bool}$	$\mathsf{iszero} \in [\{0\}]\to\mathrm{Bool}$	$\mathsf{iszero}.s = (s = [\,])$
$(+1) \in \mathrm{Nat}\to\mathrm{Nat}$	$\mathsf{suc} \in [\{0\}]\to[\{0\}]$	$\mathsf{suc}.s = 0 \vdash s$
$(-1) \in \mathrm{Nat}\curvearrowright\mathrm{Nat}$	$\mathsf{pred} \in [\{0\}]\curvearrowright[\{0\}]$	$\mathsf{pred}.(0 \vdash s) = s,$

where $\mathrm{dom}\,(-1) = \mathrm{Nat} \setminus \{0\}$ and $\mathrm{dom}\,\mathsf{pred} = [\{0\}] \setminus \{[\,]\}$. The verification of the correctness of this solution is straightforward.
□

In this example, all refinements are described in terms of the same coupling $\simeq$ (and the identities on $\{\perp\}$ and Bool). This is possible because our definition of data refinement does not refer to the types of the functions and the coupling relations. Instead of (4), an alternative definition of data refinement could be:

$$(\forall\, a, c : a\, R\, c : f.a\, S\, g.c)$$

in a context in which $R \subseteq \mathrm{dom}\, f \times \mathrm{dom}\, g$ (the context defines, for instance, $f \in A \to B$, $g \in C \to D$, and $R \subseteq A \times C$). With such a definition, the justification of the refinement of (-1) by pred requires $\simeq \setminus \{([\,], 0)\}$ as instantiation of R instead of $\simeq$, because $\simeq$ is a relation between Nat and $[\{0\}]$, not between $\mathrm{Nat} \setminus \{0\}$ and $[\{0\}] \setminus \{[\,]\}$. And for the refinement of $-$ (subtraction on the naturals) we would need another coupling. In this way we get a tailored coupling for each function that is partial with respect to the domain of the coupling and such an approach would lead to an awkward definition of algebra refinement.

Just as for algorithmic refinement, there exists a more succinct characterization of data refinement, which may be interpreted as a generalization of (3):

Property 2.10
Predicate (4) is equivalent to

(5) $\quad f \circ R \subseteq S \circ g.$

Proof

$$
\begin{array}{ll}
& (\forall\, a, c : a \in \mathrm{dom}\, f \ \wedge\ aRc : c \in \mathrm{dom}\, g \ \wedge\ f.aSg.c) \\
\equiv & \{ \text{ one-point rule (twice) } \} \\
& (\forall\, a, b, c : b = f.a \ \wedge\ a \in \mathrm{dom}\, f \ \wedge\ aRc : (\exists\, d : d{=}g.c : c{\in}\mathrm{dom}\, g \wedge bSd)) \\
\equiv & \{ \text{ property of functions } \} \\
& (\forall\, a, b, c : bfa \ \wedge\ aRc : (\exists\, d :: bSd \ \wedge\ dgc)) \\
\equiv & \{ \text{ predicate calculus: } (\forall\, x :: P(x) \Rightarrow Q) \equiv (\exists\, x :: P(x)) \Rightarrow Q \ \} \\
& (\forall\, b, c : (\exists\, a :: bfa \ \wedge\ aRc) : (\exists\, d :: bSd \ \wedge\ dgc)) \\
\equiv & \{ \text{ definition of } \circ \text{ (twice) } \} \\
& (\forall\, b, c : b(f \circ R)c : b(S \circ g)c) \\
\equiv & \{ \text{ definition of } \subseteq \} \\
& f \circ R \subseteq S \circ g.
\end{array}
$$

□

Interesting properties of data refinement are easily derived from this characterization. For example, the composition of two successive data refinements is again a data refinement:

Property 2.11 (cf. Property 2.7)

(a) $f \circ I \subseteq I \circ f$ "reflexivity"

(b) $f \circ R_0 \subseteq S_0 \circ g \;\wedge\; g \circ R_1 \subseteq S_1 \circ h \;\Rightarrow\; f \circ (R_0 \circ R_1) \subseteq (S_0 \circ S_1) \circ h$ "transitivity"

(c) $f_0 \circ R \subseteq S \circ g_0 \;\wedge\; f_1 \circ S \subseteq T \circ g_1 \;\Rightarrow\; (f_1 \circ f_0) \circ R \subseteq T \circ (g_1 \circ g_0)$ "monotonicity",

where I is an identity relation with $\mathrm{dom}\, I \supseteq \mathrm{dom}\, f \cup \mathrm{rng}\, f$.

Proof Relying on the associativity of $\circ$ and its monotonicity w.r.t. $\subseteq$ (Property 2.7d), we observe for the last two parts:

$$\begin{array}{ll} & f \circ R_0 \circ R_1 \\ \subseteq & \{\ f \circ R_0 \subseteq S_0 \circ g\ \} \\ & S_0 \circ g \circ R_1 \\ \subseteq & \{\ g \circ R_1 \subseteq S_1 \circ h\ \} \\ & S_0 \circ S_1 \circ h; \end{array} \qquad \begin{array}{ll} & f_1 \circ f_0 \circ R \\ \subseteq & \{\ f_0 \circ R \subseteq S \circ g_0\ \} \\ & f_1 \circ S \circ g_0 \\ \subseteq & \{\ f_1 \circ S \subseteq T \circ g_1\ \} \\ & T \circ g_1 \circ g_0. \end{array}$$

□

2.4 Refinement of monoalgebras

In the development of programs by stepwise refinement, one often encounters a data refinement step which comprises simultaneous substitutions of a type and a number of connected operations by some other type and corresponding operations. In order to separate such substitutions from the overall development, we introduce the notion of *algebra refinement*. In a refinement of a monoalgebra there is only one relevant coupling which we usually denote by $\simeq$.

Definition 2.12
Let A be a monoalgebra with data type A. Let C be a monoalgebra with data type C and as many operations as A. Let $\simeq$ be a relation on C and A. Then algebra A is said to be *(data-)refined* by algebra C under coupling $\simeq$ when each operation of A is data-refined by the corresponding operation of C under coupling $\simeq$, which is defined as follows for each of the three representatives of first-order operations (cf. Definitions 1.9 and 2.8).

(a) Creation $f \in T \to A$ is data-refined by creation $g \in T \to C$ under coupling $\simeq$ if

$$(\forall\, x : x \in T : f.x \simeq g.x).$$

(b) Transformation $f \in A \times A \curvearrowright A$ is data-refined by transformation $g \in C \times C \curvearrowright C$ under coupling $\simeq$ if

$$(\forall\, a,b,c,d : (a,b) \in \operatorname{dom} f \;\; \wedge \;\; a \simeq c \;\; \wedge \;\; b \simeq d : \\ (c,d) \in \operatorname{dom} g \;\; \wedge \;\; f.a.b \simeq g.c.d) \;\; .$$

(c) Inspection $f \in A \to T$ is data-refined by inspection $g \in C \to T$ under coupling $\simeq$ if

$$(\forall\, a,c : a \simeq c : f.a = g.c).$$

Algebra C is said to *implement* A when there exists a coupling $\simeq$ such that C refines A under $\simeq$.
□

As a corollary of Property 2.11 we then have

Property 2.13

(a) Algebra A is refined by A under the identity coupling on A's data type.

(b) If A is refined by C under coupling $\simeq$ and C is refined by E under coupling $\cong$, then A is refined by E under coupling $\simeq \circ \cong$.

□

The definition of algebra refinement enables us to formulate specifications of data structures very concisely. For instance, the cumbersome problem description in Example 2.9, in which the specifications of the respective operations are all very much alike, may be rendered as follows:

> Design a concrete refinement of algebra $(\,\mathrm{Nat} \mid 0, (=0), (+1), (-1)\,)$ with signature $(\,C \mid \mathsf{zero}, \mathsf{iszero}, \mathsf{suc}, \mathsf{pred}\,)$ under coupling $\simeq$.

In such specifications we use *signatures* to name the respective components of the requested refinement. A definition of an algebra assigns a value to each component of its signature in the same fashion as a definition of a function assigns a value to its name. Instead of prescribing a signature and a name for the coupling, we may also leave this open by saying:

> Design an implementation of algebra $(\,\mathrm{Nat} \mid 0, (=0), (+1), (-1)\,)$.

The specifications of zero, iszero, suc, and pred in Example 2.9 are specific instances of the general pattern implied by Definition 2.4. The following example exhibits some more "typical data refinements" involving one nontrivial coupling $\simeq$, the other couplings being identities.

Example 2.14
Let A, C and T be data types. Let $\simeq$ be a relation on C and A. We say that function f is data-refined by function g under coupling $\simeq$ when

(a) $f \simeq g$ for constants $f \in A$ and $g \in C$

(b) $(\forall\, a, c : a \simeq c : f.a \simeq g.c)$ for $f \in A \rightarrow A$ and $g \in C \rightarrow C$

(c) $(\forall\, a, c : a \in \operatorname{dom} f \;\wedge\; a \simeq c : c \in \operatorname{dom} g \;\wedge\; f.a \simeq g.c)$
for $f \in A \curvearrowright A$ and $g \in C \curvearrowright C$

(d) $(\forall\, a, c : a \in \operatorname{dom} f \;\wedge\; a \simeq c : c \in \operatorname{dom} g \;\wedge\; f.a = g.c)$
for $f \in A \curvearrowright T$ and $g \in C \curvearrowright T$.

Now, strictly speaking, (c) and (d) are conflicting: in case $A = C = T$ both are applicable but they yield different results. Sets A, C, and T are however intended to be data types of different algebras, so a possible way to prevent such conflicts is to use the algebras instead of the data types to type functions (see Remark 1.10). In this thesis we do not wish to be so rigorous because in our examples it will be clear which parts have to be refined.

Note that in (b), $a \simeq c$ implies $a \in A$ and $c \in C$, hence (b) is a special case of (c). Furthermore, note that (a) and (b) express that g is data-refined by f under coupling $\simeq^{\star}$, hence, in these cases data refinement is "symmetric". In general, however, data refinement is "asymmetric" since the refinement may have a larger domain in the sense that its domain contains values to which no values in the domain of the refined function are coupled (cf. algorithmic refinement).
□

Instead of "typical data refinements" we can also use the term "induced couplings":

Example 2.15
A coupling $\simeq$ between A and C induces couplings between for example (a) $T \times A$ and $T \times C$, (b) $[A]$ and $[C]$, and (c) $\langle [A] \rangle$ and $\langle [C] \rangle$. Such induced couplings are denoted by the same name and defined in the obvious way:

(a) $(x, a) \simeq (y, c) \;\equiv\; x = y \;\wedge\; a \simeq c$

(b) $as \simeq cs \;\equiv\; \#as = \#cs \;\wedge\; (\forall\, i : 0 \leq i < \#as : as.i \simeq cs.i)$

(c) $\langle\,\rangle \simeq \langle\,\rangle$ and $\langle la, as, ra \rangle \simeq \langle lc, cs, rc \rangle \;\equiv\; la \simeq lc \;\wedge\; as \simeq cs \;\wedge\; ra \simeq rc.$

Note that in the last case, only trees of the same shape are coupled to each other. Together with Definition 2.8, induced couplings provide an alternative way to describe "typical data refinements".

The data refinement relation between $A \rightarrow A$ and $C \rightarrow C$ can also be considered as an induced coupling (of a higher order, though). Couplings between for instance $\{A\}$ and $\{C\}$ are induced in a less straightforward way.
□

In many applications, the coupling relation is a *function* from C to A. We call such couplings *abstractions*, and we will frequently use $[\![\cdot]\!]$ instead of $\simeq$ to denote an abstraction. The coupling in Example 2.9, for instance, is an abstraction from $[\{0\}]$ to Nat, viz. $[\![s]\!] = \#s$. In general, the coupling corresponding to abstraction $[\![\cdot]\!]$ is given by $a \simeq c \;\equiv\; a = [\![c]\!]$.

Another important case is that the *dual* of a coupling is a function from A to C. We call such couplings *representations*, and we use $(\![\cdot]\!)$ instead of $\simeq^\star$ to denote these. For example, we have $(\![n]\!) = (\dashv 0)^n.[\,]$ as representation in Example 2.9. In general, the coupling corresponding to representation $(\![\cdot]\!)$ is given by $a \simeq c \;\equiv\; (\![a]\!) = c$. When describing refinements with representation functions, we will henceforth omit *: we speak of a refinement under $(\![\cdot]\!)$ instead of a refinement under $(\![\cdot]\!)^\star$, as for example in the following property.

Property 2.16 (cf. Definition 2.12)
Let C refine A under representation $(\![\cdot]\!)$, $(\![\cdot]\!) \in A \to C$ and under abstraction $[\![\cdot]\!]$, $[\![\cdot]\!] \in C \to A$, respectively. Then we have for the corresponding operations of A and C, respectively:

(a) Creations $f \in T \to A$ and $g \in T \to C$ satisfy

$$(\forall\, x : x \in T : (\![f.x]\!) = g.x),$$

and

$$(\forall\, x : x \in T : f.x = [\![g.x]\!]).$$

(b) Transformations $f \in A{\times}A{\curvearrowright}A$ and $g \in C{\times}C{\curvearrowright}C$ satisfy

$$(\forall\, a, b : (a, b) \in \operatorname{dom} f \;:\; ((\![a]\!), (\![b]\!)) \in \operatorname{dom} g \;\;\wedge\;\; (\![f.a.b]\!) = g.(\![a]\!).(\![b]\!)),$$

and

$$(\forall\, c, d : ([\![c]\!], [\![d]\!]) \in \operatorname{dom} f \;:\; (c, d) \in \operatorname{dom} g \;\;\wedge\;\; f.[\![c]\!].[\![d]\!] = [\![g.c.d]\!]).$$

(c) Inspections $f \in A \to T$ and $g \in C \to T$ satisfy

$$(\forall\, a : a \in A : f.a = g.(\![a]\!)),$$

and

$$(\forall\, c : c \in C : f.[\![c]\!] = g.c).$$

□

To keep this property simple we have confined it to total abstraction functions and total representation functions. Partial abstraction functions are particularly useful when describing pointer implementations; for example, see Section 4.3 where abstraction $[\![\cdot]\!]$ is defined only for pointers that correspond to a finite list.

Two frequently occurring instantiations of the general pattern implied by Property 2.16 are as follows.

Example 2.17
Let A and C be data types. Let $(\!|\cdot|\!) \in A \rightarrow C$. Then f is said to be data-refined by g under representation $(\!|\cdot|\!)$ when

$$(\!|f|\!) = g \qquad\qquad \text{for constants } f \in A \text{ and } g \in C$$
$$(\forall a : a \in \operatorname{dom} f \; : (\!|a|\!) \in \operatorname{dom} g \;\;\wedge\;\; (\!|f.a|\!) = g.(\!|a|\!)) \qquad \text{for } f \in A \curvearrowright A \text{ and } g \in C \curvearrowright C.$$

Let $[\![\cdot]\!] \in C \rightarrow A$. Then f is said to be data-refined by g under abstraction $[\![\cdot]\!]$ when

$$f = [\![g]\!] \qquad\qquad \text{for constants } f \in A \text{ and } g \in C$$
$$(\forall c : [\![c]\!] \in \operatorname{dom} f \; : c \in \operatorname{dom} g \;\;\wedge\;\; f.[\![c]\!] = [\![g.c]\!]) \qquad \text{for } f \in A \curvearrowright A \text{ and } g \in C \curvearrowright C.$$

□

Since many data refinements can be expressed conveniently in terms of functional couplings, one may wonder why we have introduced relational couplings at all. Well, a compelling reason is that transitivity cannot be formulated for functional data refinements, when data refinements are used under representations as well as under abstractions. For instance, a data refinement of type A by C under representation $(\!|\cdot|\!) \in A \rightarrow C$, followed by a data refinement of type C by E under abstraction $[\![\cdot]\!] \in E \rightarrow C$ is a data refinement under coupling $(\!|\cdot|\!)^\star \circ [\![\cdot]\!]$ (Property 2.13b), but $(\!|\cdot|\!)^\star \circ [\![\cdot]\!]$ is in general not a function. By restricting couplings to either representations or abstractions, transitivity is of course retained, but—as we will argue in the next section—this is too restrictive to be useful.

2.5 Surjectivity and injectivity of monoalgebras

A well-known phenomenon in the design of data structures is that a single addition to the repertoire of operations can make all the difference to the implementation. This is demonstrated in the following example.

Example 2.18
In this example Nat plays again the role of abstract type. We will successively refine the following algebras:

$$\begin{aligned} \mathsf{Nuts0} &= (\,\mathrm{Nat} \mid (+1)\,) \\ \mathsf{Nuts1} &= (\,\mathrm{Nat} \mid (+1), 0\,) \\ \mathsf{Nuts2} &= (\,\mathrm{Nat} \mid (+1), 0, (=0)\,) \\ \mathsf{Nats} &= (\,\mathrm{Nat} \mid (+1), 0, (=0), (-1)\,), \end{aligned}$$

by a concrete algebra with data type C under coupling $\simeq \subseteq C \times \mathrm{Nat}$, such that the size of C is minimal in each refinement.

We start with Nuts0. An immediate consequence of the definition of data refinement (cf. (2.14b)) is that $\emptyset$ refines $(+1)$ under coupling $\emptyset$. Hence, in this case we may take $C = \emptyset$. This result is to be expected because a data refinement of a single operation of type $A \rightarrow A$ is pointless, since there is no way to provide this operation with an argument. For example, we cannot use Nuts0 to refine $(+1).0$, since we have no corresponding data refinement of 0.

Nuts1 provides 0 as operation. To refine 0 we see that we cannot take C empty anymore (cf. (2.14a)). It is, however, still possible to refine $(+1)$ and 0 as follows: take as concrete type $\{0\}$ and as coupling $\{(0,n) \mid n \in \mathrm{Nat}\}$; then the respective refinements are $\{(0,0)\}$ and 0. Hence, in this case C is a singleton. This result is not strange either because, although it is possible to generate any natural number with operations 0 and $(+1)$, there is no way to distinguish any two distinct naturals. So, it is not surprising that it is possible to represent all natural numbers by the same value.

The situation changes slightly when we consider Nuts2. Again, however, we are able to refine these operations using a finite concrete type. Operationally speaking, $(=0)$ may be used to distinguish 0 from the positive naturals, but there is no way to distinguish distinct positive naturals. We may therefore choose $\{0,1\}$ as concrete type and $\{(0,0)\} \cup \{(1,n{+}1) \mid n \in \mathrm{Nat}\}$ as coupling. The operations of Nuts2 are refined by $\{(0,1),(1,1)\}$, 0, and $(=0)$, respectively.

Finally, by adding operation (-1) the concrete type cannot be finite anymore: we use a unary representation in which a list of n zeros represents natural n.

The respective refinements are summarized in the following table.

	C	suc	zero	iszero	pred
Nuts0	$\emptyset$	$\mathsf{suc} = \emptyset$			
Nuts1	$\{0\}$	$\mathsf{suc}.0 = 0$	$\mathsf{zero} = 0$		
Nuts2	$\{0,1\}$	$\mathsf{suc}.0 = 1$ $\mathsf{suc}.1 = 1$	$\mathsf{zero} = 0$	$\mathsf{iszero}.b = b{=}0$	
Nats	$[\{0\}]$	$\mathsf{suc}.s = s \dashv 0$	$\mathsf{zero} = [\,]$	$\mathsf{iszero}.s = s{=}[\,]$	$\mathsf{pred}.(s \dashv 0) = s$

For Nuts0, coupling $\simeq$ is the empty set; for the other refinements, couplings $\simeq$ are given by $n \simeq 0$, $0 \simeq 0$ and $n{+}1 \simeq 1$, and $\#s \simeq s$, respectively.

All these refinements can also be described as refinements under a representation. For Nuts0, representation $(\!|\cdot|\!)$ is the empty set, which is a partial function from Nat to C. For the other refinements, representation $(\!|\cdot|\!)$ is a total function of type $\mathrm{Nat} \rightarrow C$, with, respectively, $(\!|n|\!) = 0$, $(\!|0|\!) = 0$ and $(\!|n{+}1|\!) = 1$, and $(\!|n|\!) = (\dashv 0)^n.[\,]$. However, only the refinements of Nuts0 and Nats can be described as refinements under an abstraction $[\![\cdot]\!] \in C \rightarrow \mathrm{Nat}$, viz. $[\![\cdot]\!] = \emptyset$ and $[\![s]\!] = \#s$. The other refinements cannot be refinements under an abstraction, since the existence of abstraction $[\![\cdot]\!]$ implies that C has to be infinite: C cannot be empty, for $\mathsf{zero} \in C$; by induction, $[\![\mathsf{suc}^n.\mathsf{zero}]\!] = n$, hence $(\forall\, m, n : m \neq n : \mathsf{suc}^m.\mathsf{zero} \neq \mathsf{suc}^n.\mathsf{zero})$.

By interchanging the roles of abstract and concrete we see that there are also refinements that cannot be described as refinements under a representation.
□

From this example we learn that there are refinements that cannot be described by abstractions or representations. Algebras Nuts1 and Nuts2 are, however, a bit artificial and therefore we will present some more realistic examples shortly; in these examples it is even impossible to use functional couplings. The fact that there exist interesting nonfunctional couplings is another compelling reason to use relational couplings in Definition 2.8.

Yet, many designs of data structures can be modelled as refinements under an abstraction function—often even a *surjective* one. In the sequel we investigate under which circumstances this is the case. An algebra is called *injective* when all its refinements can be described by means of abstractions. To prepare for a formal definition of injectivity we first introduce the notion of *surjectivity*.

Definition 2.19
For monoalgebra A, the set of "reachable" values is defined as the largest subset X of A's data type satisfying

$$(\forall\, a : a \in X : (\exists\, c :: a \simeq c)),$$

for all algebras C and relations $\simeq$, for which C refines A under coupling $\simeq$. We call X the *range* of A and denote it by $\mathrm{rng}\,\mathsf{A}$. A monoalgebra is called *surjective* when its range equals its data type.
□

The values outside the range of an algebra need not be represented in an implementation. Operationally speaking, the range of an algebra contains all values of its data type that can be generated by any composition of its operations.

Example 2.20

$$\begin{array}{lcl}
\mathrm{rng}\,(\,\mathrm{Nat} \mid 0\,) & = & \{0\} \\
\mathrm{rng}\,(\,\mathrm{Nat} \mid (+1)\,) & = & \emptyset \\
\mathrm{rng}\,(\,\mathrm{Nat} \mid 0, (+1)\,) & = & \mathrm{Nat} \\
\mathrm{rng}\,(\,\mathrm{Nat} \mid 0, (+2)\,) & = & \{2n \mid n \in \mathrm{Nat}\} \\
\mathrm{rng}\,(\,\mathrm{Nat} \mid 0, 1, (+2)\,) & = & \mathrm{Nat} \\
\mathrm{rng}\,(\,\mathrm{Nat} \mid 0, (*2), (+1) \circ (*2)\,) & = & \mathrm{Nat} \\
\mathrm{rng}\,(\,[T] \mid [\,], \vdash\,) & = & [T].
\end{array}$$

Although the equalities in this example will be intuitively clear, a proof requires some effort. Let us prove the middle one.

Let $\mathsf{N2}$ denote algebra $(\,\mathrm{Nat} \mid 0, (+2)\,)$. We first remark that $\mathsf{N2}$ is refined by $(\,\mathrm{Nat} \mid 0, (+1)\,)$ under coupling $\simeq$ given by $2n \simeq n$. So, there exists a refinement of $\mathsf{N2}$ in which no odd natural is coupled to a concrete value. This implies $\mathrm{rng}\,\mathsf{N2} \subseteq \{2n \mid n \in \mathrm{Nat}\}$.

To prove the other inclusion $\mathrm{rng}\,\mathsf{N2} \supseteq \{2n \mid n \in \mathrm{Nat}\}$, we reason as follows. Suppose $(\,C \mid \mathsf{zero}, \mathsf{suc2}\,)$ refines $\mathsf{N2}$ under coupling $\simeq$. Then we prove $(\forall\, n :: (\exists\, c :: 2n \simeq c))$ by induction on n.

Case $n = 0$. Since zero refines 0, we conclude that $0 \simeq \mathsf{zero}$.

Case $n = m{+}1$. We derive

$$
\begin{array}{ll}
 & (\exists\, c :: 2m \simeq c) \\
\Rightarrow & \{\ \mathsf{suc2} \text{ refines } (+2) \text{ under } \simeq\ \} \\
 & (\exists\, c :: 2m{+}2 \simeq \mathsf{suc2}.c) \\
\Rightarrow & \{\ \text{dummy transformation } c := \mathsf{suc2}.c\ \} \\
 & (\exists\, c :: 2m{+}2 \simeq c) \\
\equiv & \{\ n = m{+}1\ \} \\
 & (\exists\, c :: 2n \simeq c).
\end{array}
$$

□

So, by considering surjective algebras only, some strange ones are eliminated. However, an algebra like $(\,\mathrm{Nat} \mid 0, (+1)\,)$ is still a bit strange. Although all naturals can be generated with 0 and $(+1)$, there is no way to distinguish any two distinct naturals. We introduce the notion of *injectivity* to capture this phenomenon.

Definition 2.21
A monoalgebra A is called *injective* when

$$(\forall\, c :: (\#\, a : a \in \mathrm{rng}\,\mathsf{A}\ : a \simeq c) \leq 1),$$

for all algebras C and relations $\simeq$, for which C refines A under coupling $\simeq$.
□

Note that an algebra is injective when its range is empty. The operational interpretation is that an algebra is injective when any two distinct values in its range can be distinguished by some composition of its operations; in other words, there should exist compositions that produce different results for distinct values.

Example 2.22

$(\,\mathrm{Nat} \mid (+1)\,)$	is injective
$(\,\mathrm{Nat} \mid 0, 1, (=0)\,)$	is injective

$(\,\mathrm{Nat} \mid 0, (+1), (=0)\,)$	is not injective
$(\,\mathrm{Nat} \mid 0, (+2), (-2), (=0)\,)$	is injective
$(\,\mathrm{Nat}\backslash\{0\} \mid 1, (*2), (\mathrm{div}\,2), (=1)\,)$	is injective
$(\,\mathrm{Real} \mid e, \pi, nth\,)$	is injective (see Example 1.4)
$(\,[\{0\}] \mid [\,], (=[\,]), \vdash, tl\,)$	is injective
$(\,[\mathrm{Bool}] \mid [\,], (=[\,]), \vdash, tl\,)$	is not injective
$(\,[T] \mid [\,], (=[\,]), \vdash, hd, tl\,)$	is injective, provided T belongs to an injective algebra.

To give an example of a proof of injectivity we demonstrate that Nats from Example 2.18 is injective.

Let $(\,C \mid \mathsf{suc}, \mathsf{zero}, \mathsf{iszero}, \mathsf{pred}\,)$ refine Nats under coupling $\simeq$. Since Nats is surjective we have to show that $(\forall\, c :: (\#\, n :: n \simeq c) \leq 1)$. To this end, we observe for any c and for any m and n, $m \leq n$:

$$
\begin{array}{ll}
 & m \simeq c \;\wedge\; n \simeq c \\
\Rightarrow & \{\ \mathsf{pred} \text{ refines } (-1) \text{ under } \simeq\ \} \\
 & m{-}m \simeq \mathsf{pred}^m.c \;\wedge\; n{-}m \simeq \mathsf{pred}^m.c \\
\Rightarrow & \{\ \mathsf{iszero} \text{ refines } (=0) \text{ under } \simeq\ \} \\
 & 0 = 0 \;\equiv\; \mathsf{iszero}.(\mathsf{pred}^m.c) \;\wedge\; n{-}m = 0 \;\equiv\; \mathsf{iszero}.(\mathsf{pred}^m.c) \\
\Rightarrow & \{\ \text{predicate calculus}\ \} \\
 & m = n.
\end{array}
$$

□

In the above proof of injectivity of Nats, we have directly applied Definition 2.21. Another way to show this result is by programming $=$ on Nat in terms of the operations of Nats:

Property 2.23 (=-test)
Consider a surjective monoalgebra A with data type A. If $=$ of type $A{\times}A \rightarrow \mathrm{Bool}$ can be expressed in terms of the operations of A, then algebra A is injective.

Proof Let C refine A under coupling $\simeq$. Define **eq** in terms of the operations of C in the same way as $=$ is expressed in terms of the operations of A. Then **eq** refines $=$ under $\simeq$. That is:

$$(\forall\, a, b, c, d : a \simeq c \;\wedge\; b \simeq d : a = b \;\equiv\; \mathbf{eq}.c.d).$$

To show that A is injective, suppose that c is coupled to both a and b, that is suppose that $a \simeq c$ and $b \simeq c$. Instantiation of the above quantification then yields $a = b \;\equiv\; \mathbf{eq}.c.c$. Since $\mathbf{eq}.c.c$ is, for instance, also equivalent to $a = a$, which is equivalent to true, we conclude that $a = b$.
□

Remark 2.24
The algebras in Example 1.3 are surjective but not injective because they do not provide any inspection operations: all operations involve type Bool (and $\{\bot\}$), but for an inspection we need an extra type. According to our definition of algebra refinement, a possible implementation of $(\,\text{Bool} \mid \text{false}, \neg \circ \wedge\,)$ is therefore $(\,\{0\} \mid 0, \{((0,0),0)\}\,)$. This implementation is however not very useful and to exclude it we have to add an inspection operation to the algebra of booleans, say $(=\text{false}) \in \text{Bool} \to \{F, T\}$. Here, $\{F,T\}$ is the data type of a both surjective and injective algebra, and we simply have to postulate the existence of this algebra. We cannot describe an implementation of such an algebra other than by assuming the existence of another algebra of this type; in the end, we need to postulate the existence of such an algebra (as "hardware").
□

Note that omission of "$a \in \text{rng}\,\mathsf{A}$" from Definition 2.21 yields that algebra $(\,\text{Nat} \mid 0, 1, (=0)\,)$ is not injective. Indeed, omitting $a \in \text{rng}\,\mathsf{A}$ turns about every nonsurjective algebra into a noninjective one, as the reader may verify.

From the results in Examples 2.20 and 2.22 it follows that Nats is bijective:

Definition 2.25
A monoalgebra A is called *bijective* when it is both surjective and injective, that is, when

$$(\forall\, a :: (\#\, c :: a \simeq c) \geq 1) \;\wedge\; (\forall\, c :: (\#\, a :: a \simeq c) \leq 1),$$

for all algebras C and relations $\simeq$, for which C refines A under coupling $\simeq$.
□

The importance of this notion is that any refinement of a bijective algebra can be described as a data refinement under a (surjective) abstraction function. The conclusion of our investigation is that we can safely use abstractions in specifications of injective algebras, without running the risk of excluding efficient refinements that can only be justified using a representation function or a coupling relation (as in Example 2.18).

Unfortunately, it is not always obvious that an algebra is bijective. For example, for positive m and n, algebra $(\,\text{Nat} \mid 0, (+m), (-n), (=0)\,)$ has range $\{k \;\text{gcd}(m,n) \mid k \in \text{Nat}\}$, the multiples of $\text{gcd}(m,n)$. Hence, it is surjective precisely when $\text{gcd}(m,n) = 1$, and it is unconditionally injective.

What is more, the removal of an operation from an algebra can turn it from injective into noninjective, as shown in the following example.

Example 2.26
We add operation $\downarrow$ to the algebra of queues of Example 1.5:

$$(\,[\text{Int}] \mid [\,], (=[\,]), \dashv, hd, tl, \downarrow\,).$$

Since the algebra of queues is bijective, this algebra is also bijective, and we can use abstraction functions to describe its refinements. However, in some applications operation hd is not required, and then the following algebra will do:

$$([\mathrm{Int}] \mid [\,], (=[\,]), \dashv, tl, \downarrow).$$

For this algebra a more efficient implementation is designed in Chapter 8, exploiting the fact that it is not injective (e.g., $[1,0]$ and $[2,0]$ are indistinguishable). Since different abstract values are represented by the same concrete value, the efficient implementation cannot be described by an abstraction function.
□

We hope that we have convinced the reader that it is necessary to introduce the notion of data refinement under a coupling relation, although many interesting refinements can be modelled in terms of abstraction functions. For obviously-bijective algebras it is best to use abstractions in a specification of a data structure, but when this is not so clear—or, when one does not want to investigate this—it is better to use a coupling relation.

2.6 A closer look at surjectivity and injectivity

As argued in Example 2.26, noninjective algebras arise naturally as specifications of data structures. Efficient implementations of these data structures, however, exploit the fact that indistinguishable abstract values can be represented by the same concrete value. The following theorem shows that this is always possible—and also that unreachable values need not be represented (in case of nonsurjective algebras).

Theorem 2.27 Any monoalgebra has a bijective refinement. □

This theorem follows from the two lemmas below and the transitivity of algebra refinement (Property 2.13b).

Lemma 2.28 Any monoalgebra has a surjective refinement.

Proof Let A be a monoalgebra with data type A. We define a surjective refinement R of A as follows. The data type of R is $\mathrm{rng}\,\mathsf{A}$, and for each operation f of A, the corresponding operation f' of R is defined as (see Definition 2.12):

(a) $f' = f$, if f is a creation of type $T \to A$.

(b) $f' = f{\restriction}(\mathrm{rng}\,\mathsf{A} \times \mathrm{rng}\,\mathsf{A})$, if f is a transformation of type $A \times A \curvearrowright A$.

(c) $f' = f{\restriction}(\mathrm{rng}\,\mathsf{A})$, if f is an inspection of type $A \to T$.

The coupling $\simeq$ between these algebras is the identity on $\text{rng}\,\mathsf{A}$.

Now, observe that $\text{rng}\,\mathsf{A}$ is a subset of A, hence $\simeq$ is a relation on $\text{rng}\,\mathsf{A}$ and A. To prove that R refines A under $\simeq$, we show that f' refines f under $\simeq$ in the respective cases.

Case (a). We have to prove, cf. Definition 2.12a:

$$(\forall\, x : x \in T : f.x \simeq f'.x).$$

Since $f' = f$ and $\simeq$ is the identity on $\text{rng}\,\mathsf{A}$, it suffices to prove that $f.x \in \text{rng}\,\mathsf{A}$, for all $x \in T$. On account of the definition of rng this means that in any refinement of A, $f.x$ should be represented by a concrete value. This is indeed the case, since by the definition of data refinement $g.x$ represents $f.x$ for any refinement g of f.

Case (b). We have to prove, cf. Definition 2.12b:

$$\begin{aligned}(\forall\, a,b,c,d : (a,b) \in \text{dom}\, f \;\wedge\; a \simeq c \;\wedge\; b \simeq d :\\ (c,d) \in \text{dom}\, f' \;\wedge\; f.a.b \simeq f'.c.d).\end{aligned}$$

Since $a \simeq c \;\equiv\; a \in \text{rng}\,\mathsf{A} \;\wedge\; a = c$ and $\text{dom}\, f' = \text{dom}\, f \cap (\text{rng}\,\mathsf{A} \times \text{rng}\,\mathsf{A})$, the interesting part left to prove is that $f.a.b \in \text{rng}\,\mathsf{A}$ for $(a,b) \in \text{dom}\, f'$. That is, we have to show that $f.a.b$ is represented in any refinement of A. So, let g refine f. Since $a \in \text{rng}\,\mathsf{A}$ there is a representation c, say, of a. Similarly, there exists a representation d of b. By the definition of data refinement $(c,d) \in \text{dom}\, g$, hence application of g to these values yields $g.c.d$ as representation of $f.a.b$.

Case (c). We have to prove, cf. Definition 2.12c:

$$(\forall\, a,c : a \simeq c : f.a = f'.c).$$

Since $a \simeq c \;\equiv\; a \in \text{rng}\,\mathsf{A} \;\wedge\; a = c$, this is equivalent to $(\forall\, a : a \in \text{rng}\,\mathsf{A} : f.a = f'.a)$, which follows from the definition of f'.

To show that R is surjective, we prove $\text{rng}\,\mathsf{A} \subseteq \text{rng}\,\mathsf{R}$ as follows. By the transitivity of algebra refinement it follows that any refinement of R under coupling $\cong$, say, is a refinement of A under $\simeq \circ \cong$. The latter coupling equals $\cong$ because $\simeq$ is the identity on $\text{rng}\,\mathsf{A}$. So, any refinement of R is also a refinement of A under the same coupling. Therefore, any value in the range of A is in the range of R.

□

Lemma 2.29 Any surjective monoalgebra has a bijective refinement.

Proof Let A be a surjective monoalgebra with data type A. Binary relation $\sim$ on A is defined by $a \sim b$ if there exists a refinement of A in which a and b are represented by the same value. This relation is symmetric, hence the reflexive-transitive closure of $\sim$, denoted as $\overset{*}{\sim}$, is an equivalence relation.

Using this equivalence relation, we define a bijective refinement E of A as follows. Its data type E consists of the equivalence classes of $\overset{*}{\sim}$. Using $(\!|\cdot|\!)$ to denote the induced mapping from A to E, we define for each operation f of A, the corresponding operation f' of E by:

(a) $f'.x = (\!|f.x|\!)$ for $x \in T$, if f is a creation of type $T \to A$.

(b) $f'.(\!|a|\!).(\!|b|\!) = (\!|f.a.b|\!)$ for $(a,b) \in \operatorname{dom} f$, if f is a transformation of type $A \times A \curvearrowright A$.

(c) $f'.(\!|a|\!) = f.a$ for $a \in A$, if f is an inspection of type $A \to T$.

By Property 2.16, f' refines f under $(\!|\cdot|\!)$, provided f' is a well-defined function. This proviso is indeed fulfilled, as we will now prove for each case.

Case (a). This case is trivial, since $f' = (\!|\cdot|\!) \circ f$.

Case (b). In this case, well-definedness means that $(\!|f.a.b|\!) = (\!|f.a'.b'|\!)$ if $(\!|a|\!) = (\!|a'|\!)$ and $(\!|b|\!) = (\!|b'|\!)$, for (a,b) and (a',b') in $\operatorname{dom} f$. Or, in terms of $\overset{*}{\sim}$: $a \overset{*}{\sim} a' \wedge b \overset{*}{\sim} b' \Rightarrow f.a.b \overset{*}{\sim} f.a'.b'$. To this end we first prove that $a \sim a' \Rightarrow f.a.b \sim f.a'.b$. So, assume $a \sim a'$. Then, by the definition of $\sim$, there exists a refinement in which a and a' are represented by the same value c, say. Since A is surjective, b is also represented, say by d. Let g be the refinement of f, then $(c,d) \in \operatorname{dom} g$ and $g.c.d$ represents both $f.a.b$ and $f.a'.b$, hence $f.a.b \sim f.a'.b$. Interchanging the roles of a and b gives a similar property. Together these properties suffice to complete the proof by an induction argument, which we omit.

Case (c). For an inspection f, we prove $a \overset{*}{\sim} a' \Rightarrow f.a = f.a'$. This follows by induction from $a \sim a' \Rightarrow f.a = f.a'$, which is proved as follows. Assumption $a \sim a'$ gives that there exists a refinement in which a and a' are represented by the same value c, say. Let g refine f. Then, by definition of data refinement (2.12c), $g.c$ is equal to both $f.a$ and $f.a'$, hence $f.a = f.a'$.

So much for the proof that E refines A under $(\!|\cdot|\!)$. We now show that E is bijective.

To see that E is surjective, we consider an element $(\!|a|\!)$ of E. Any refinement of E under coupling $\simeq$ also refines A under $(\!|\cdot|\!)^\star \circ \simeq$ (Property 2.13b). This implies, on account of the surjectivity of A, that there exists a representation c of a in such a refinement. But this means that $a \, (\!|\cdot|\!)^\star \circ \simeq c$. Or, $(\!|a|\!) \simeq c$, hence c represents $(\!|a|\!)$.

Finally, to see that E is injective, assume that $(\!|a|\!)$ and $(\!|b|\!)$ are represented by the same value c in some refinement C of E under a coupling $\simeq$. Then, by the transitivity of algebra refinement, $a \, (\!|\cdot|\!)^\star \circ \simeq c$ and $b \, (\!|\cdot|\!)^\star \circ \simeq c$. Hence, C is a refinement of A in which a and b are coupled to the same value c. Therefore $a \sim b$, hence $(\!|a|\!) = (\!|b|\!)$.
□

Remark 2.30
Since E refines A under representation $(\!|\cdot|\!)$, E identifies elements a and b of A whenever $a \overset{*}{\sim} b$. This implies that $\overset{*}{\sim}$ and $\sim$ are actually the same relations on A. We have used $\overset{*}{\sim}$ in the proof because a direct proof of the transitivity of $\sim$ is somewhat awkward.
□

Chapter 3

Amortized complexity

To amortize: to end (a debt) by making regular payments into a special fund.
[Oxford advanced learner's dictionary]

In a traditional worst-case analysis of an algorithm one derives a worst-case bound by adding the worst-case bounds of the constituent parts of the algorithm (which are often operations on data structures). However, bounds obtained in this fashion may be overly pessimistic, possibly orders of magnitude too high. Amortization is a simple principle to analyze the worst-case behaviour of algorithms in a compositional way: to avoid overly pessimistic bounds, so-called *amortized* cost measures are used instead of *actual* cost measures for the components of algorithms.

As is often the case with simple mathematical principles, there are many ways to formulate them and to explain them. Sometimes a principle is so simple that it is often applied without even knowing its name, let alone its existence (e.g., the Pigeonhole Principle). It is however important to expose such principles in a convenient way and to collect instructive applications of them. In imperative programming, for example, the Invariance Theorem is a coding of the Principle of Mathematical Induction particularly suited for the design of repetitions (see, e.g., [5]). In this chapter we will present several views of amortization based on the banker's and physicist's views of Sleator and Tarjan [31]. Of course, simple as such principles can be, their application will still be difficult when the problem is inherently complex. (It is, for instance, an open problem whether pairing heaps [7] are as efficient as Fibonacci heaps [8] in the amortized sense.)

In the next section, we explain the idea of amortization in an imperative setting. In the subsequent section, we present abstract versions of the banker's and physicist's views of amortization, and we show that they are equally powerful. The subject of Chapter 5 is a calculational approach to the (amortized) analysis of functional programs, in particular operations of algebras. The material in this chapter provides a basis for that approach.

3.1 Amortized analysis of imperative programs

As a simple imperative programming language we use Dijkstra's guarded command language (GCL) [5]. A suitable cost measure for GCL programs is the number of times that guards of repetitions are passed. Apart from constant factors, the time complexity of these programs is then determined.

To explain how amortization is applied in the analysis of GCL programs, we consider the following nondeterministic program (with $N \geq 0$):

$$
\begin{array}{l}
m, n := 0, 0 \\
\{ \text{ invariant: } 0 \leq m \ \wedge \ 0 \leq n \leq N;\ \text{bound: } m + 2(N{-}n) \ \} \\
;\ \mathbf{do}\ n \neq N \rightarrow m, n := m{+}1, n{+}1 \\
\quad [\!]\ \ m \neq 0 \rightarrow m := m{-}1 \\
\ \ \mathbf{od}\ .
\end{array}
$$

This program occurs as common projection of many programs that employ, say, a stack as auxiliary variable. Variable m then corresponds to the height of such a stack, and, among other steps, these programs repeatedly either *push* a value onto the stack ($m := m{+}1$), or *pop* a value from the stack ($m := m{-}1$). Initially and upon termination the stack is empty. Instead of stacks, other data structures that are often used in this fashion are first-in first-out queues (with put and get operations), priority queues (with insertion of an arbitrary value and removal of the minimum value), etc.

Remark 3.1
Usually, those programs consist of a nested repetition, followed by a repetition that empties the data structure. For example:

$$
\begin{array}{l}
\ \ m, n := 0, 0 \\
;\ \mathbf{do}\ n \neq N\ \rightarrow \\
\qquad \mathbf{do}\ \ldots\ \rightarrow \{m \neq 0\}\ m := m{-}1\ \mathbf{od} \\
\qquad ;\ m, n := m{+}1, n{+}1 \\
\ \ \mathbf{od} \\
;\ \mathbf{do}\ m \neq 0\ \rightarrow m := m{-}1\ \mathbf{od}.
\end{array}
$$

The number of steps of the inner repetition per step of the outer repetition depends on more variables than m and n alone. This irregular behaviour is nicely captured by the nondeterminism in the former program, which we prefer for its simplicity.
□

There are many ways to prove that the repetition of the above program is unfolded exactly $2N$ times, or even that both alternatives are unfolded N times each. An informal way to do this is as follows. It is obvious that there are exactly N pushes. Moreover, there corresponds one pop to each push, since the stack is empty initially and upon termination. Therefore, the number of pops is also exactly N. A more formal way to prove that the repetition is unfolded $2N$ times is by means of a bound function (see program annotation). The bound function $m + 2(N{-}n)$ is decremented in each step. Since, its initial value is $2N$ and its final value is 0, there are $2N$ steps (hence, the program terminates).

We consider both of the above ways inadequate for the purpose of *analyzing* GCL programs. The first approach because it is informal and ad hoc, the second because it also entails a termination proof which we consider part of the correctness proof of a program. In general, we analyze only correct (hence terminating) programs.

Given that the program terminates, a formal, but simpler way to analyze it is as follows. We introduce a fresh variable t to count the number of unfoldings of the repetition and establish an invariant of the form $t = E$:

$$
\begin{array}{l}
m, n := 0, 0;\ t := 0 \\
\{\ \text{invariant: } t = 2n - m\ \} \\
;\ \mathbf{do}\ n \neq N \rightarrow m, n := m{+}1, n{+}1;\ t := t{+}1 \\
\quad [\!]\ \ m \neq 0 \rightarrow m := m{-}1;\ t := t{+}1 \\
\ \ \mathbf{od}\ \ .
\end{array}
$$

From the annotation in this program we conclude that $t = 2N$ holds as post-condition. Note that we count one unit of cost for each stack-operation, except for the initialization of the stack (initialization $t := 1$ gives a more realistic measure).

The problem with the last approach is that we have to invent an exact relation for t, viz. $t = 2n - m$. Although this relation is not that complicated, it is in many cases difficult to find an exact relation. Often we must content ourselves with an upper bound for t. As we will explain below, the principle of amortization helps to achieve a better separation of concerns in such analyses. A key point in the following amortized analysis is that we exploit the fact that it is clear that the first alternative of the repetition is executed exactly N times (and that it is not so clear that the second alternative is also executed exactly N times).

In addition to variable t, the *(accumulated) actual* costs, we introduce variable a, the *(accumulated) amortized* costs. We see to it that upon termination $a = t$ by taking 2 as amortized costs for a push and 0 as amortized costs for a pop. This *choice* for the amortized costs is guided by the observation that the accumulated number of pushes is simply equal to n, while an expression for the accumulated number of pops is more complicated (viz. $n - m$). In this way, we obtain a simple relation for the amortized costs, viz. $a = 2 * n + 0 * (n{-}m)$:

$$
\begin{array}{l}
m, n := 0, 0;\ t, a := 0, 0 \\
\{\ \text{invariant: } a = t + m\ \wedge\ a = 2n\ \} \\
;\ \mathbf{do}\ n \neq N \rightarrow m, n := m{+}1, n{+}1;\ t, a := t{+}1, a{+}2 \\
\quad [\!]\ \ m \neq 0 \rightarrow m := m{-}1;\ t := t{+}1 \\
\ \ \mathbf{od}\ \ .
\end{array}
$$

The analysis can now be divided into two parts, each part corresponding to a conjunct of the above invariant. The first part shows that upon termination $a = t$, that is, the amortized costs are equal to the actual costs in the end. The second part gives a simple relation for the amortized costs. Proving each of the parts is simpler than proving the invariance of $t = 2n - m$ in one go, but taken together the amortized analysis is of course more laborious. However, in more complicated applications of amortization this degree of disentanglement is really fruitful.

Instead of defining the amortized costs of each step of a repetition directly (which corresponds to the *banker's view* of amortization), there is also an indirect way (corresponding to the *physicist's view*). Then the amortized costs are defined in terms of a *potential function*, a function defined on the state space of the program (cf. a bound function in the Invariance Theorem). The potential of a state can be thought of as the accumulated difference between the amortized costs and the actual costs. The potential function, often called Φ, corresponding to the choice of amortized costs in the above analysis is m:

$$
\begin{array}{l}
m, n := 0, 0;\ t := 0 \\
\{\ \text{invariant: } t + \Phi = 2n;\ \text{potential } \Phi\text{: } m\ \} \\
;\ \mathbf{do}\ n \neq N \rightarrow m, n := m{+}1, n{+}1;\ t := t{+}1 \\
\quad [\,]\ \ m \neq 0 \rightarrow m := m{-}1;\ t := t{+}1 \\
\ \ \mathbf{od}\ \ .
\end{array}
$$

In general, $t + \Phi$ equals the accumulated amortized costs, and the change in this quantity per step equals the amortized cost of a step. Notice that the potential m is nonnegative, which means that there is always a surplus: the accumulated actual costs do not exceed the accumulated amortized costs.

Potential m ("size of stack, queue, etc.") is a natural potential function. However, since the initial value of m equals its final value, any real-valued function of m will do, for example $\Phi = -m$ or $\Phi = 0$. In the first case, the amortized costs of a push are zero and a pop costs two units, so we count the pops. This approach is not so good when the number of pops is possibly less than the number of pushes; then some pushes are never counted. The second case gives an analysis in which the amortized costs equal the actual costs, so this does not help. Some potentials even lead to *negative* amortized costs (like $\Phi = 2m$, which gives -1 as amortized cost for a pop, and 3 for a push), or to *nonconstant* amortized costs (like $\Phi = m^2$).

In general, we choose an invariant of the form $t + \Phi = E$, and we choose potential Φ such that a simple expression E for the amortized costs results. Often it is convenient to have a nonnegative potential, since this guarantees that $t \leq E$ at any point in the program. Sometimes we must content ourselves with an upper bound for the amortized costs. The form of the invariant is then $t + \Phi \leq E$, as in the following program. This program has been obtained from the previous program by substituting N for $m{+}1$.

$$
\begin{array}{l}
m, n := 0, 0;\ t := 0 \\
\{ \text{ invariant: } 0 \leq m \ \wedge \ t + \Phi \leq n(N{+}1); \text{ potential } \Phi\text{: } m \ \} \\
;\ \mathbf{do}\ n \neq N \rightarrow m, n := N, n{+}1;\ t := t{+}1 \\
\quad [\!]\ \ m \neq 0 \rightarrow m := m{-}1;\ t := t{+}1 \\
\ \ \mathbf{od}\ \ .
\end{array}
$$

It follows that $t \leq N^2 + N$ upon termination (which is a tight bound).

We conclude this section with a more complicated application of the "potential technique".

Example 3.2
Kaldewaij derives in [20, Section 12.3] the following program for the computation of the length of a shortest segment of array X in which the maximum value on that segment occurs exactly twice:

$$
\begin{array}{l}
\ \ n, r := 1, \infty \\
;\ \mathbf{do}\ n \neq N \ \rightarrow \\
\qquad s := n{-}1 \\
\qquad ;\ \mathbf{do}\ s \neq 0 \ \wedge \ X[s] < X[n] \ \rightarrow s := f[s]\ \mathbf{od} \\
\qquad ;\ f[n] := s \\
\qquad ;\ \mathbf{if}\ X[s] = X[n] \ \rightarrow r := r \min (n{+}1{-}s) \\
\qquad\ \ [\!]\ X[s] \neq X[n] \ \rightarrow \text{skip} \\
\qquad\ \ \mathbf{fi} \\
\qquad ;\ n := n{+}1 \\
\ \ \mathbf{od}\ \ .
\end{array}
$$

Here, integer array X of length N, $N{\geq}1$, is the input array and the output is stored in integer variable r.

The problem with the analysis of this program is that the number of steps of the inner repetition per step of the outer repetition is difficult to determine. Instead, we therefore perform an amortized analysis in which we determine the *total* number of steps of the inner repetition.

To this end we concentrate on the variables n, f, and s. For the analysis, the relevant invariant for the outer repetition reads:

$$
\text{P}: \quad 0 < n \ \wedge \ (\forall\, i : 0 < i < n : 0 \leq f[i] < i),
$$

hence statement $s := f[s]$ decreases the value of s and proper termination of the inner repetition is guaranteed. To avoid counting the number of steps of this repetition per step of the outer repetition, we choose the potential such that the amortized costs per step of the inner repetition are *zero*. That is, since a step of the inner repetition consists of $s := f[s]$, we take as potential $\Psi = \#s$, with

$$
\begin{array}{lcl}
\#0 & = & 0 \\
\#i & = & 1 + \#(f[i])
\end{array}
$$

for $0 < i < n$. The invariance of $0 \le s < n \ \wedge\ \mathrm{P}$ guarantees that this potential is well-defined.

The amortized analysis is now completed by showing that the following annotation is correct, using that the invariance of P (and $0{\le}s{<}n$ for the inner repetition) has already been shown in [20]. We use one potential for each repetition:

$$
\begin{array}{l}
\quad n := 1;\ t := 0 \\
\{\ \text{invariant: P} \ \wedge\ t + \Phi = n{-}1;\ \text{potential } \Phi\text{: } \#(n{-}1)\ \} \\
;\ \mathbf{do}\ n \ne N\ \ \rightarrow \\
\qquad s := n{-}1 \\
\quad\ \ \{\ \text{invariant: } 0 \le s < n \ \ \wedge\ \text{P} \ \ \wedge\ t + \Psi = n{-}1;\ \text{potential } \Psi\text{: } \#s\ \} \\
\quad\ \ ;\ \mathbf{do}\ s \ne 0\ \wedge\ \ldots\ \rightarrow s := f[s];\ t := t{+}1\ \mathbf{od} \\
\quad\ \ ;\ f[n] := s \\
\quad\ \ ;\ n := n{+}1 \\
\quad \mathbf{od}\ \ .
\end{array}
$$

The definition of potential Φ has been obtained by applying the substitution rule for $s := n{-}1$ to the definition of Ψ. The invariance of $t + \Psi = n{-}1$ for the inner repetition is trivial, and it is matter of straightforward verification to show the invariance of $t + \Phi = n{-}1$:

$$
\begin{array}{ll}
 & ((t + \#(n{-}1) = n - 1)^{n}_{n+1})^{f}_{f[n:=s]} \\
\equiv & \{\ \text{see definition of } \#' \text{ below}\ \} \\
 & t + \#'n = n \\
\equiv & \{\ \#'n = 1 + \#'s = 1 + \#s, \text{ since } 0 \le s < n\ \} \\
 & t + \#s = n - 1,
\end{array}
$$

where $\#'$ is given by

$$
\begin{array}{lcl}
\#'0 & = & 0 \\
\#'i & = & 1 + \#'(f[n{:=}s][i])
\end{array}
$$

for $0{<}i{<}n{+}1$. The last line of this derivation may be written as $t + \Psi = n{-}1$, which is a postcondition of the inner repetition.

Since Φ is nonnegative, we have that $t \le N{-}1$ upon termination. Hence the program is linear in N, since the outer repetition is unfolded exactly $N{-}1$ times.
□

3.2 Abstract views of amortization

In [31, Section 2], Tarjan describes two views of amortization, called the banker's view and the physicist's view, and he states that these views are "entirely equivalent". There is however no precise framework to support the meaning of this

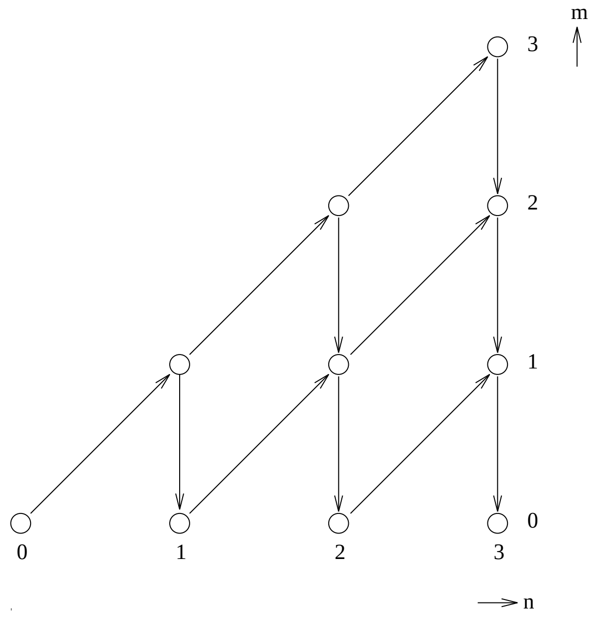

Figure 3.1: An abstract algorithm with $s = (0,0)$.

statement, let alone to justify it. Based on Tarjan's description, Mehlhorn and Tsakalidis also mention two views of amortization in [24, p.304] but here the equivalence is trivial: they speak of "a function *bal* which maps the possible configurations of the data structure into the real numbers" when describing the banker's view, but this is precisely the idea of a potential function.

What is not pointed out in [24] and what is not clear in Tarjan's explanation is that the essential point about potential functions is that they depend on the *current state* (of the data structure) only, whereas a more general notion of amortization (based on the banker's view) may take the entire *history* (of the data structure) into account. By recording the history of the data structure, it is of course possible to simulate a banker's analysis by a physicist's analysis, but this we consider pointless. In the sequel we will show how a banker's analysis may be simulated by a physicist's analysis *without* extending the state of the data structure.

Before we present an abstract view of amortization, we introduce an abstract view of algorithms.

Definition 3.3
An *abstract algorithm* is a quadruple (V, E, s, t), in which V is a set of *states*, $E \subseteq V \times V$ (*steps*), $s \in V$ is the *initial state*, and $t \in E \to \text{Real}$ (*(actual) costs per step*). Furthermore, all states should be *reachable* from s, where the graph terminology applies to the directed graph (V, E).
□

Note that it is not required that V (or E) is finite. Neither it is required that (V, E) is acyclic, or that vertices in (V, E) should have a finite number

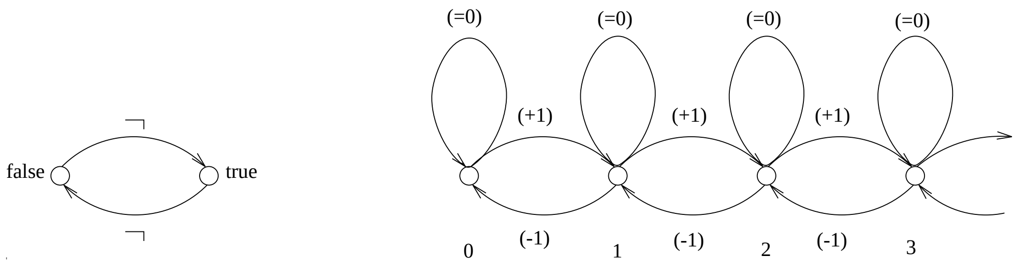

Figure 3.2: Abstract algorithms corresponding to algebras $(\,\text{Bool} \mid \text{false}, \neg\,)$ and $(\,\text{Nat} \mid 0, (=0), (+1), (-1)\,)$.

of predecessors and successors. An example of an abstract algorithm is given in Figure 3.1; it corresponds to all possible executions of the program in the previous section for $N{=}3$. The cost function t is the all-one function in this case.

One may ask why we do not exclude cyclic graphs in the above definition, since at first sight this corresponds to non-terminating programs. However, the state of an abstract algorithm does not necessarily correspond to the *entire* state of a program; it may also correspond to a *projection* of the state. For instance, projecting the diagram in Figure 3.1 on variable m gives a cyclic abstract algorithm. What is more, as depicted in Figure 3.2, algebras typically correspond to cyclic abstract algorithms in which the initial state is reachable from all states.

The fact that an abstract algorithm has only one initial state is not really a limitation. For example, algebras with several constants can be viewed as abstract algorithms with initial state $\perp$ and the respective constants as successors of $\perp$. Furthermore, if there are several transitions between two states with different costs, this can be modelled in an abstract algorithm as a single step whose cost is the largest cost of these transitions. (If this is not satisfactory one must distinguish these transitions by distinguishing more states.) Another issue is that algebras often provide binary operators, like addition of natural numbers. In such circumstances one cannot just take a single natural value as state (cf. Figure 3.2) but, for instance, a *bag* of naturals is required. This phenomenon is particularly relevant for implementations of *mergeable* priority queues (Chapter 9) and for implementations of similar algebras that are efficient in the amortized sense only.

Now that we have defined an abstract notion of algorithms, we are ready to introduce an abstract notion of amortization. As shown in the previous section, an amortized analysis of an imperative program consists of a choice of the amortized costs for each step of the program such that for all paths from the initial state to the final state(s), the actual costs are bounded from above by the amortized costs of these paths. Since an abstract algorithm does not necessarily have final states (states without successors), we have to be a little bit more careful:

Definition 3.4
An *abstract amortized analysis* is a quintuple (V, E, s, t, a), in which (V, E, s, t) is an abstract algorithm and $a \in E \to \text{Real}$ (*amortized costs per step*). Furthermore, for each vertex x there should exist a vertex y reachable from x, for which the actual cost of every *finite* path[1] from the initial state s to y is at most its amortized cost.
□

We call $a(x, y) - t(x, y)$ the *surplus* of edge (x, y). The surplus of a path is the sum of the surpluses of the composing edges. A *pluspoint* is a vertex for which the surpluses of all paths from s ending in it are nonnegative. By definition, each vertex has a pluspoint within its reach. Note that a cycle cannot have a negative surplus, for this would lead to unbounded negative surpluses for paths from s to vertices on the cycle; this contradicts with the requirement that every vertex has a pluspoint within its reach. Also note that the initial state is a pluspoint, because the empty path has surplus zero and all cycles ending in the initial state have nonnegative surpluses.

We now distinguish a special kind of amortization, in which the amortized costs are defined in terms of a potential function. This corresponds to the physicist's method of Sleator and Tarjan [31].

Definition 3.5
An abstract amortized analysis (V, E, s, t, a) is called *conservative*, when there exists a function $\Phi \in V \to \text{Real}$ such that $a(x, y) = t(x, y) + \Phi.y - \Phi.x$. Such a function is called a *potential (function)*.
□

Note that the surplus of edge (x, y) equals potential difference $\Phi.y - \Phi.x$. Furthermore, any cycle has surplus zero in a conservative analysis. The pluspoints of a conservative analysis are those states x for which $\Phi.x \geq \Phi.s$.

The physicist's method seems less powerful but we will show that it is as powerful as the banker's method ("as powerful as" in the sense of the theorem below). The problem with the physicist's method is that the amortized costs of a step have to be defined in terms of a function on the *vertices*, whereas in the banker's method amortized costs can be allocated separately to each *edge*.

Theorem 3.6
Let $A = (V, E, s, t, a)$ be an abstract amortized analysis. Then there exists a potential function Φ, for which the conservative analysis $A' = (V, E, s, t, a')$ of the same abstract algorithm, with $a'(x, y) = t(x, y) + \Phi.y - \Phi.x$, satisfies

(a) A' has the same set of pluspoints as A, and

(b) for each step, the amortized costs in A' are at most the amortized costs in A.

[1] Throughout this section, paths are finite.

Proof The definition of Φ is as follows: for each $x \in V$, $\Phi.x$ is defined as the *infimum* of the surpluses of all paths from s to x. Before we show that (a) and (b) hold, we must first convince ourselves that Φ is indeed a function from V to Real, that is $\Phi.x \neq -\infty$ for all $x \in V$. For this we use that A is an amortized analysis, which means according to Definition 3.4 that for any state x, there exists a pluspoint y reachable from x by a path with surplus δ, say. Since y is a pluspoint, all paths from s to y have a nonnegative surplus. Hence all paths from s to x have a surplus that is bounded from below by $-\delta$. So, $\Phi.x \geq -\delta$.

Now, it is easy to show (a). First we observe that in A' the surpluses of the paths from s to a state x are all equal to $\Phi.x$. So the pluspoints in A' are those states for which the potential is nonnegative. But this means, by the definition of Φ, that the surpluses of all paths from s to x in A are nonnegative, hence that x is a pluspoint in A. (Note, in particular, that s is a pluspoint in both A and A'.)

Finally, (b) means that $a'(x, y) \leq a(x, y)$ must hold for each edge (x, y). By the definition of a' above, this may be written as $\Phi.y \leq \Phi.x + a(x, y) - t(x, y)$. This is clearly true because $\Phi.y$ is at most the surplus of any path from s to x followed by (x, y).
$\square$

The intuition behind the definition of Φ is that the *least* surplus determines the potential of a state. It is easy to see that Φ is nonnegative whenever all paths starting in s have nonnegative surpluses. Of course, when all surpluses are bounded from below by a negative constant C, say, then we may also obtain a nonnegative potential by increasing Φ by $-C$. It is however important to note that the infimum of the potentials of all states may be $-\infty$, since the surplus over *all* paths starting in s is not bounded from below. Because of this fact it is not always possible to use a nonnegative potential in an amortized analysis (see Chapter 7).

It should be noted that the proof of the above theorem does not give a nice description of the potential. To obtain the potential of a state we have to determine an infimum over all paths leading to that state. Fortunately, paths with cycles can be ignored, for adding cycles to a path cannot decrease the surplus. For the example program in the previous section, potential m corresponds to the potential defined in the theorem. In practice, the banker's method can be used when defining a potential becomes too complicated.

Chapter 4

Implementation aspects

In this chapter, we introduce the program notation that we will use to describe implementations of algebras. According to Definition 2.1, a program notation defines the borderline between abstract and concrete programs. Usually, the borderline is chosen such that concrete programs can be translated relatively easily into machine code by a compiler program. The compilation of the programs presented in this thesis is, however, of no concern to us. For our purposes it is important that the program notation is a well-chosen compromise between a high-level mathematical notation and a low-level machine language: a mathematically-oriented notation facilitates the correctness proof of programs, whereas a machine-oriented notation facilitates the definition of realistic cost measures for programs. Relying on some common knowledge of functional and imperative languages (and their implementations), we will confine ourselves to an informal description of the notation.

In favour of the mathematical part, we shall start with a *purely-functional* program notation. This goes well with the fact that we are primarily interested in monoalgebras—whose operations are functional, after all. We will use this notation in a restricted way only, so that *eager evaluation* suffices as simple and efficient evaluation method for our programs. For instance, the notation is used in a *first-order* fashion only, which means that there will be a clear distinction between the data on the one hand and the programs operating on the data on the other hand. In this way, realistic cost measures are easily defined for our programs.

In addition to the usual algebras provided by functional languages, we will also use algebras involving arrays and pointers—as provided by most imperative languages. For reasons of efficiency, operations of these algebras are implemented *destructively*. As a consequence, the usage of these algebras must be restricted to what will be called *linear usage* (cf. [36]) so as to guarantee that they behave as specified. To facilitate the use of arrays and pointers, the program notation will be extended with some "imperative features". This is illustrated by a number of pointer implementations for list algebras like stacks and queues.

4.1 Functional program notation

Our functional program notation is based on the notation of [17], which in turn finds its roots in SASL [33]. We confine ourselves to an informal introduction of the notation, which will suffice to understand the programs in the sequel. Program *Mergesort* on page 60 shows (almost) all of the features of the notation described below.

In our notation, a functional program is a function definition of the form $f.x = F$, in which f is the function's name, x is the name of the parameter and F is the defining expression. Here, expression F may refer to x, but also to f; in the latter case, the definition is recursive. The function's domain is defined informally or left implicit.

The expression in the right-hand side of a function definition is built from the operations of a number of predefined algebras. As simple algebras, we use the data types Bool, Nat, and Int, along with the usual operations. Furthermore, the algebra of stacks is assumed to be available which provides a "kernel" for the list operations (cf. Example 1.5). A similar algebra involving binary trees is also assumed to be concrete (cf. Section 1.1.4):

$$(\,\langle T\rangle \mid \langle\,\rangle, (=\langle\,\rangle), \langle\cdot,\cdot,\cdot\rangle, l, m, r\,),$$

where operation $\langle\cdot,\cdot,\cdot\rangle \in \langle T\rangle{\times}T{\times}\langle T\rangle \to \langle T\rangle$ is used to construct nonempty trees; it satisfies $\langle l.x, m.x, r.x\rangle = x$ for nonempty x. (The standard pointer implementations of these algebras are presented in Section 4.3.)

To support case analysis, we use *conditional expressions*. The *alternatives* of a conditional expression are separated by $[\!]$'s. For example, a frequently occurring form is $(E, B\ [\!]\ F, C)$, in which B and C are boolean expressions, called the *guards*. Usually, B and C exclude each other because they are complementary, but we will sometimes write programs that violate this rule in order to retain symmetry. For example, we prefer to leave a program like

$$\begin{array}{lcll} f.\langle x,a,y\rangle & = & x & ,a \le 0 \\ & [\!] & y & ,a \ge 0 \end{array}$$

nondeterministic; strengthening one of its guards with $a \neq 0$ is "left to the reader".

To support modular design, expressions may contain so-called *where-clauses*. A where-clause consists of one or more function definitions enclosed by the pair $|[$ and $]|$. Where-clauses are often used to abbreviate subexpressions that occur more than once; for instance, we write $x * x |[x = a * b - a/b]|$ instead of $(a * b - a/b) * (a * b - a/b)$.

In the left-hand side of a function definition it is allowed to use *patterns* instead of names as parameters. Patterns are restricted forms of expressions, which are useful in shortening a function definition. For instance, using the patterns $2{*}a$ and $2{*}a{+}1$, integer division by 2 may be defined by $d.(2{*}a) = a$

and $d.(2{*}a{+}1) = a$. Patterns are particularly useful in definitions of operations on structures like lists and trees (see also Chapter 6). Instead of using the pattern (x, y) when the domain of a function is a cartesian product of the form $X \times Y$, we usually write $f.x.y$ as abbreviation for $f.(x, y)$, and so on.

Merely for the sake of brevity, we will also use $\circ$ (*function composition*) in our programs. For example, a definition like $h.x = g.(f.x)$ is abbreviated to $h = g \circ f$. Because of this restricted use, it is no problem that $\circ$ is a higher-order function.

The remaining features of our program notation are supposed to be well-understood. In Section 4.3 the program notation will be extended with some imperative features. Programs written in the notation introduced in the present section will be called *purely-functional* programs.

4.2 Eager evaluation

In this section we consider a number of aspects related to the execution of functional programs. A basic understanding of these aspects is assumed in the performance analyses in later chapters. We emphasize that many other aspects, such as "lazy evaluation", "infinite lists", and "input/output processing", are ignored, simply because they do not play a role in the programs considered in this thesis.

Central to the execution of functional programs is the notion of *reduction* (or *evaluation*) of expressions. In fact, the execution of a program f, given an input value x from the domain of f, boils down to the reduction of expression $f.x$. More generally, execution of the programs considered in this thesis gives rise to the reduction of expressions of the form $f.E$. For these programs the following simple reduction method suffices to reduce $f.E$: first E is reduced, and subsequently f is applied to the *value* of E. This reduction method is known as *eager evaluation* (see, e.g., [27]), and it is applied until the expression is free of function applications. The result of the reduction is then the *value* of the expression—where we assume that the reduction indeed terminates. Here is a simple example of a reduction (cf. Example 1.4):

$$
\begin{array}{ll}
 & \mathsf{suc}.(\mathsf{suc}.\mathsf{zero}) \\
= & \{ \text{ unfold definition of } \mathsf{zero} \text{ } \} \\
 & \mathsf{suc}.(\mathsf{suc}.[\,]) \\
= & \{ \text{ unfold definition of } \mathsf{suc} \text{ } \} \\
 & \mathsf{suc}.[1] \\
= & \{ \text{ unfold definition of } \mathsf{suc}\text{, using } [1]{=}[\,] \dashv 1 \text{ } \} \\
 & \mathsf{suc}.[\,] \dashv 0 \\
= & \{ \text{ unfold definition of } \mathsf{suc}\text{, using } [1] \dashv 0{=}[1,0] \text{ } \} \\
 & [1,0],
\end{array}
$$

in which we have focused on the *unfoldings* of (the definitions of) zero and suc. The reduction of suc.(suc.zero) takes place in the context which defines zero and suc. This is common practice, and in the sequel an expression is always understood to have a context that defines its constituents.

In the next chapter, we discuss the analysis of the time complexity of functional programs. Since we will concentrate on cost measures that count unfoldings of user-defined functions, we have to motivate that those cost measures are indeed realistic. Therefore, we discuss the execution of our functional programs in more detail.

For instance, an important fact is that all operations of the predefined algebras mentioned in Section 4.1 have $O(1)$ time complexity. Furthermore, we rely on the fact that one unfolding of a user-defined function definition $f.x = F$ takes $O(1)$ time, since it merely amounts to the substitution of the value for x in expression F; the result then has to be reduced further, which may give rise to more unfoldings. For conditional expressions, we assume that only the relevant parts of the expression are evaluated. The evaluation of $(E, B \, [\!] \, F, \neg B)$, for instance, comprises one evaluation of B, followed by the evaluation of either E or F, depending on the value of B.

In the presence of where-clauses of the form $|[x = E]|$, it is important that multiple evaluation of E is avoided. The value of E should be *shared* by all occurrences of x in the expression to which the where-clause is attached. So, although $x = E$ may be written in the general form $x.\bot = E$ of a function definition, evaluation of these definitions proceeds differently for reasons of efficiency. For example, evaluation of $x + x|[x = E]|$ consists of a single evaluation of E, followed by an unfolding of $+$. However, a where-clause of the form $|[f.x = E]|$ attached to an expression F gives rise to as many unfoldings of f as there are applications of f in F, since these applications will in general be quite different.

So much for the time complexity of our programs. As for space complexity, we assume the existence of an ideal *garbage collector*, which recycles memory cells as soon as they become "garbage", thereby minimizing the maximal number of cells ever required during the execution of a program. Space complexity is particularly relevant for algebras with structured data types. In the next section we present a pointer implementation for stacks, in which each application of $\vdash$ occupies an extra cell, and an application of tl possibly releases a cell, namely when the application removes the last reference to the cell. For instance, evaluation of $tl.(a \vdash s)$ will first occupy an extra cell and then the application of tl will release this cell again because it removes the only reference to this cell.

4.3 Pointer implementation of stacks

In Example 1.8 we presented an algebra for pointers. In that algebra, the memory is explicitly represented by an element of type $\Omega \curvearrowright T$. In pointer notations of imperative languages, however, the memory is usually not explicitly mentioned.

In the example below, we will introduce a Pascal-like notation for pointers in which the memory is also left implicit.

The example concerns the standard representation for finite lists used by implementations of functional languages. For fixed type T, we consider the algebra of stacks:

$$\mathsf{S} = (\, [T] \mid [\,], (=[\,]), \vdash, hd, tl \,),$$

for which we give a refinement at pointer level with signature:

$$(\, L \mid \mathsf{empty}, \mathsf{isempty}, \mathsf{cons}, \mathsf{hd}, \mathsf{tl} \,).$$

To allow for an efficient implementation of this algebra, a list of type $[T]$ is represented by a so-called *singly-linked list.* That is, type L is defined by

$$L = {}^\wedge\langle a{:}T, r{:}L\rangle.$$

This definition of L expresses that elements of type $L\backslash\{\text{nil}\}$ are pointers to pairs of type $T\times L$. Moreover, it implies that $p^\wedge = \langle p^\wedge.a, p^\wedge.r\rangle$, for $p \in L\backslash\{\text{nil}\}$. The abstraction function is given by

$$\begin{array}{lcl} [\![\text{nil}]\!] & = & [\,] \\ [\![p]\!] & = & p^\wedge.a \vdash [\![p^\wedge.r]\!] \quad , p \neq \text{nil}, \end{array}$$

and to guarantee that the range of $[\![\cdot]\!]$ consists of *finite* lists only, the domain of $[\![\cdot]\!]$ is defined by

$$\text{dom}\,[\![\cdot]\!] \;= \{p \mid p \in L \;\wedge\; (\exists\, i : 0 \leq i : p({}^\wedge.r)^i = \text{nil})\}.$$

Remark 4.1
The above is an example of the use of partial abstraction functions. In this case we have defined the domain of $[\![\cdot]\!]$ explicitly, but often the domain is clear from the context: since $[\![\cdot]\!]$ is a partial function with range $[T]$, it follows that, for instance, a pointer p satisfying $p = p^\wedge.r$ cannot be in the domain of $[\![\cdot]\!]$; the domain of $[\![\cdot]\!]$ is the *largest* subset of L for which the above definition of $[\![\cdot]\!]$ defines values in $[T]$.
□

To facilitate the manipulations of pointers in functional programs, we extend the program notation as follows. Instead of an expression, the right-hand side of a function definition is allowed to be an imperative *statement.* This statement defines the function value by means of a call to procedure *return*; each execution of such a statement should give rise to exactly one call to return. The notation used for statements is supposed to be self-evident; procedures *new* and *dispose* are used in a Pascal-like manner in these statements.

Using this notation, the stack operations may be programmed as follows:

$$\begin{array}{lcl}
\mathsf{empty} & = & \mathrm{nil} \\
\mathsf{isempty}.p & = & p = \mathrm{nil} \\
\mathsf{cons}.a.p & = & |[\mathrm{var}\ h{:}L;\ \mathrm{new}(h);\ h^\wedge := \langle a, p\rangle;\ \mathrm{return}(h)\ ]| \\
\mathsf{hd}.p & = & p^\wedge.a \\
\mathsf{tl}.p & = & p^\wedge.r.
\end{array}$$

Note that the program for cons is not functional: the value of $\mathsf{cons}.a.p$ is not determined by the values of a and p, because operation new is not a function. However, the value of $(\mathsf{cons}.a.p)^\wedge$ is a function of a and p, and this is the value that matters: $[\![p]\!]$ depends on the value of $p^\wedge$ only, for $p \neq \mathrm{nil}$.

In Section 4.1 we have assumed that the following algebra involving binary trees is also part of the functional program notation:

$$\mathsf{T} = (\ \langle T\rangle \ \mid\ \langle\ \rangle, (=\langle\ \rangle), \langle\cdot,\cdot,\cdot\rangle, l, m, r\ ).$$

This algebra is in essence the same as algebra S, and therefore we confine ourselves to a brief description of its implementation. The appropriate pointer type is

$$B = {}^\wedge\langle l{:}B, a{:}T, r{:}B\rangle,$$

for which we have as corresponding abstraction:

$$\begin{array}{lcl}
[\![\mathrm{nil}]\!] & = & \langle\ \rangle \\
[\![p]\!] & = & \langle [\![p^\wedge.l]\!], p^\wedge.a, [\![p^\wedge.r]\!]\rangle \quad , p \neq \mathrm{nil}.
\end{array}$$

With this representation each of the operations of T can be supported in $O(1)$ time.

4.4 Destructivity

Mainly for two reasons, purely-functional programming languages do not provide arrays and pointers. The first and minor reason is that the underlying algebras for arrays and pointers involve nonfunctional operations, viz. operation ? for arrays (cf. Example 1.7) and operation *new* for pointers (cf. Example 1.8).

The second and major reason is, however, that *efficient* implementations of array and pointer operations are *destructive*. For instance, evaluation of $a[i{:=}x]$ in general destroys the representation of a because the value of $a[i]$ is simply overwritten so as to achieve $O(1)$ time complexity. To guarantee that these operations behave in accordance with their specifications, the use of these operations must be restricted. For instance, an expression like $(a[0{:=}11], a[0{:=}13])$ should be avoided because evaluation of this expression results either in $(a[0{:=}11], a[0{:=}11])$ or in $(a[0{:=}13], a[0{:=}13])$, depending on which component of the pair is evaluated first.

Destructivity is related to the fact that parameters of structured types such as arrays and lists are passed *by reference* rather than *by value*. A pass-by-reference mechanism has to be used for these types to achieve the desired time complexity. For instance, to achieve $O(1)$ time complexity for an operation that operates on structures of size N, a pass-by-value mechanism cannot be used, for this already requires $O(N)$ time to copy the parameter.

Yet, destructivity is not a necessary consequence of the use of a pass-by-reference mechanism. For instance, in the implementation of stacks in the previous section none of the operations is destructive, although all list parameters are passed by reference ("reference p is passed instead of value $p^\wedge$"). Phrased differently, all parameters can be thought of as being passed by value, although they are actually passed by reference. Therefore, this implementation of stacks can be used in implementations of purely-functional programming languages. Only after addition of dispose(p) to the program for $\mathsf{tl}.p$ is a destructive implementation obtained:

$$\mathsf{tl}.p \quad = \quad \mathrm{return}(p^\wedge.r);\ \mathrm{dispose}(p).$$

This particular implementation of stacks has the advantage that cells are recycled explicitly so that no garbage collection is required. The price to be paid is that, for instance, $(q,q)\ |[q = \mathsf{tl}.p]|$ and $(\mathsf{tl}.p, \mathsf{tl}.p)$ are not equivalent anymore: evaluation of the former expression yields two references to $\mathsf{tl}.p$, whereas evaluation of the latter one fails because evaluation of either of the two occurrences of $\mathsf{tl}.p$ disposes of the cell to which p points, and this blocks the evaluation of the remaining occurrence. To prevent such problems, this implementation may be used in a *linear* fashion only, as will be explained in Section 4.6.

4.5 Queues and concatenable deques

For fixed type T, we now present destructive implementations for two more algebras operating on lists of type $[T]$. In these implementations we shall use well-known techniques such as circularly-linked lists and doubly-linked lists. The operations of these implementations all have $O(1)$ time complexity.

First, we consider the algebra of queues:

$$\mathsf{Q} = (\,[T]\ \mid\ [\,], (=[\,]), hd, tl, \dashv\,),$$

for which we give two destructive refinements at pointer level. We use the following signature:

$$(\,Q\ \mid\ \mathsf{empty}, \mathsf{isempty}, \mathsf{hd}, \mathsf{tl}, \mathsf{snoc}\,).$$

In our first refinement, two pointers are used, one of them pointing to the head of a singly-linked list and the other one to the last cell in that list (if present). More precisely, we take $Q = L{\times}L$, with $L = {}^\wedge\langle a{:}T, r{:}L\rangle$ (as in Section 4.3), and we take as coupling

$$s \simeq \langle p, q\rangle \;\equiv\; s = list.p \;\wedge\; (s \neq [\,] \Rightarrow q = p(^\wedge.r)^{\#s-1}),$$

where

$$\begin{array}{lcll} list.\mathrm{nil} & = & [\,] & \\ list.p & = & p^\wedge.a \vdash list.(p^\wedge.r) & , p \neq \mathrm{nil}. \end{array}$$

Note that $s = list.p$ implies $p(^\wedge.r)^{\#s} = \mathrm{nil}$. This coupling leads to the following programs:

$$\begin{array}{lcl}
\mathsf{empty} & = & \langle \mathrm{nil}, \mathrm{nil}\rangle \\
\mathsf{isempty}.\langle p, q\rangle & = & p = \mathrm{nil} \\
\mathsf{hd}.\langle p, q\rangle & = & p^\wedge.a \\
\mathsf{tl}.\langle p, q\rangle & = & \mathrm{return}(\langle p^\wedge.r, q\rangle);\ \mathrm{dispose}(p) \\
\mathsf{snoc}.\langle p, q\rangle.a & = & |[\mathrm{var}\ h{:}L;\ \mathrm{new}(h);\ h^\wedge := \langle a, \mathrm{nil}\rangle \\
 & & ;\ (\ \ \mathrm{return}(\langle h, h\rangle) \qquad\qquad\quad ,\ p = \mathrm{nil} \\
 & & \quad [\!]\ \ q^\wedge.r := h;\ \mathrm{return}(\langle p, h\rangle)\ \ ,\ p \neq \mathrm{nil} \\
 & & \quad) \\
 & &]|.
\end{array}$$

Note that removal of dispose(p) from the program for tl does *not* turn this implementation into a nondestructive one, because evaluation of $\mathsf{snoc}.\langle p, q\rangle.a$ mutilates the list represented by $\langle p, q\rangle$.

The above implementation requires $2+\#s$ pointers to represent queue s. Another implementation that uses $1+\#s$ pointers is obtained by using so-called *circularly-linked lists*. The idea is to exploit the fact that, according to the above coupling, pointer $q^\wedge.r$ is equal to nil when $p \neq \mathrm{nil}$. To that end, the value of p is stored in $q^\wedge.r$, that is, we take $Q = L$ and we define the coupling by $[\,] \simeq \mathrm{nil}$ and for nonempty s:

$$s \simeq q \;\equiv\; \#s = (\mathrm{Min}\, i : 1 \le i : q(^\wedge.r)^i{=}q) \;\wedge\; s = list.(q^\wedge.r)\!\uparrow\!\#s.$$

Hence, q points to the representation of the last element of s. The programs are:

$$\begin{array}{lcl}
\mathsf{empty} & = & \mathrm{nil} \\
\mathsf{isempty}.q & = & q = \mathrm{nil} \\
\mathsf{hd}.q & = & q^\wedge.r^\wedge.a \\
\mathsf{tl}.q & = & |[\mathrm{var}\ p{:}L;\ p := q^\wedge.r \\
 & & ;\ (\ \ q := \mathrm{nil} \qquad\quad ,\ p = q \\
 & & \quad [\!]\ \ q^\wedge.r := p^\wedge.r\ \ ,\ p \neq q \\
 & & \quad) \\
 & & ;\ \mathrm{return}(q);\ \mathrm{dispose}(p) \\
 & &]|
\end{array}$$

$$
\begin{array}{ll}
\mathsf{snoc}.q.a = & |[\mathrm{var}\ h{:}L;\ \mathrm{new}(h) \\
 & ;\ (\ \ h^\wedge := \langle a, h\rangle \qquad\qquad\qquad ,\ q = \mathrm{nil} \\
 & \quad [\!]\ \ h^\wedge := \langle a, q^\wedge.r\rangle;\ q^\wedge.r := h\ \ ,\ q \neq \mathrm{nil} \\
 & \quad) \\
 & ;\ \mathrm{return}(h) \\
 &]|.
\end{array}
$$

The use of circularly-linked lists pays off when, for example, N queues with a total length of N are to be represented; then circularly-linked lists require only $2N$ pointers, whereas singly-linked lists require $3N$ pointers.

It is left to the reader to verify that both pointer representations for queues allow $O(1)$ implementations for operations $+\!\!+$ and lt ("last"). However, extending the repertoire of operations with operation ft ("front") requires a change of representation to achieve the desired degree of efficiency. The algebra that supports these additional operations is called the algebra of *concatenable deques* ("concatenable double-ended queues"):

$$\mathsf{CDQ} = (\,[T]\ \mid\ [\,], (=[\,]), [\cdot], +\!\!+, hd, tl, lt, ft\,).$$

As signature for the refinement of CDQ we use

$$(\,D\ \mid\ \mathsf{empty}, \mathsf{isempty}, \mathsf{single}, \mathsf{cat}, \mathsf{hd}, \mathsf{tl}, \mathsf{lt}, \mathsf{ft}\,).$$

To accommodate manipulations at both ends of lists of type $[T]$, we now use *doubly-linked lists* to represent these. That is, pointer type D is defined by

$$D = {}^\wedge\langle l{:}D, a{:}T, r{:}D\rangle.$$

Like a pointer of type L, a pointer p of type D represents list $list.p$. To enable an efficient implementation of ft, the coupling is defined as:

$$s \simeq p\ \equiv\ s = list.p\ \wedge\ (\forall\, i : 0 \le i < \#s : p(^\wedge.r)^i = p(^\wedge.l)^{\#s-i}).$$

Note that $p(^\wedge.l)^{\#s} = p$ for $p \neq \mathrm{nil}$, hence the l-links form a circular list and $p^\wedge.l$ points to the cell corresponding to the last element of $list.p$. Note also that $p(^\wedge.r)^{\#s} = \mathrm{nil}$, hence the representation is not symmetric in l and r. Of course, we could have chosen for a representation symmetric in l and r but, since we want to use only one pointer that either points to the first or to the last cell in the doubly-linked list, the symmetry is broken anyway. The corresponding programs are also asymmetric:

$$
\begin{array}{lcl}
\mathsf{empty} & = & \mathrm{nil} \\
\mathsf{isempty}.p & = & p = \mathrm{nil} \\
\mathsf{single}.a & = & |[\mathrm{var}\ h{:}D;\ \mathrm{new}(h);\ h^\wedge := \langle h, a, \mathrm{nil}\rangle;\ \mathrm{return}(h)\ ]| \\
\mathsf{cat}.p.q & = & q \hfill ,p = \mathrm{nil} \\
 & [\!] & p \hfill ,q = \mathrm{nil} \\
 & [\!] & p^\wedge.l^\wedge.r := q;\ p^\wedge.l, q^\wedge.l := q^\wedge.l, p^\wedge.l;\ \mathrm{return}(p) \\
 & & \hfill ,p \neq \mathrm{nil}\ \wedge\ q \neq \mathrm{nil} \\
\mathsf{hd}.p & = & p^\wedge.a
\end{array}
$$

$$
\begin{array}{lcll}
\mathsf{tl}.p & = & \text{nil} & ,p^\wedge.r = \text{nil} \\
 & [] & p^\wedge.r^\wedge.l := p^\wedge.l;\ \text{return}(p^\wedge.r);\ \text{dispose}(p) & ,p^\wedge.r \neq \text{nil} \\
\mathsf{lt}.p & = & p^\wedge.l^\wedge.a & \\
\mathsf{ft}.p & = & \text{nil} & ,p^\wedge.r = \text{nil} \\
 & [] & |[\text{var } h{:}D;\ h := p^\wedge.l;\ p^\wedge.l := h^\wedge.l & \\
 & & ;\ h^\wedge.l^\wedge.r := \text{nil};\ \text{return}(p);\ \text{dispose}(h) & \\
 & &]| & ,p^\wedge.r \neq \text{nil}\ .
\end{array}
$$

Clearly, this implementation is destructive.

4.6 Linear usage of destructive monoalgebras

We call an algebra destructive if at least one of its operations is destructive. To guarantee the correctness of programs in which destructive algebras are used, certain conditions must be satisfied. Such restrictions are formally described by P. Wadler in [36]. Using his terminology, the use of destructive algebras must be *linear*. Roughly speaking, this means that in a program no multiple references ("pointers") are created to variables that are subject to destructive operations.

In this thesis, the idea of linearity is explained informally and illustrated by a number of examples. For a more precise description we refer to [36]. We discuss restrictions on the use of x in function definitions of the form $f.x = F$, where the data type of x belongs to a destructive monoalgebra. As a first approximation, linear usage of x means in this case that

> each evaluation of F should give rise to at most one application of a *transformation* on x.

Hence, the use of creations and inspections is not restricted by this rule. Such a transformation is either a transformation of the algebra to which x belongs or a user-defined one. For instance, the identity on the data type of x is a user-defined transformation, which implies that definition $dup.x = (x, x)$ is not allowed. We say that this definition contains a *fork in* x, since (x, x) contains *two* applications of the identity function. Clearly, when x is used linearly there are no forks in x, and vice versa. Note that conditional expressions may contain more than one transformation on x. For instance, $(x, B \mathbin{[]} x, \neg B)$ contains two transformations, but its evaluation gives rise to only one of these.

We illustrate the notion of linearity by some examples of linear and nonlinear usage of the destructive implementation of stacks:

$$
\begin{array}{lcl}
\mathsf{empty} & = & \text{nil} \\
\mathsf{isempty}.p & = & p = \text{nil} \\
\mathsf{cons}.a.p & = & |[\text{var } h{:}L;\ \text{new}(h);\ h^\wedge := \langle a, p\rangle;\ \text{return}(h)\ ]| \\
\mathsf{hd}.p & = & p^\wedge.a \\
\mathsf{tl}.p & = & \text{return}(p^\wedge.r);\ \text{dispose}(p).
\end{array}
$$

Clearly, the use of tl must be restricted because it is destructive. The use of the other transformation cons is also restricted because evaluation of cons.$a.p$ creates an extra reference to p (and to a). There are no restrictions for creation empty and inspections isempty and hd.

To show how these operations can be used in a correct way, we consider the program:

$$f_1.p = \mathsf{cons}.(\mathsf{hd}.p).(\mathsf{tl}.p).$$

Exploiting the fact that eager evaluation does not prescribe the order in which the arguments of cons are to be evaluated, hd.p can be evaluated before tl.p. Hence, there exists a *safe* evaluation order so that execution of this program indeed establishes $[\![f_1.p]\!] = [\![p]\!]$—as desired. Since tl.$p$ destroys p, function f_1 is also destructive.

A safe evaluation order does not exist when two or more destructive operations are to be performed on the same variable, as in

$$f_2.p = \mathsf{cons}.(\mathsf{hd}.p).(\mathsf{cons}.(\mathsf{hd}.(\mathsf{tl}.p)).(\mathsf{tl}.(\mathsf{tl}.p))).$$

Evaluation of either the first or the second occurrence of tl.p destroys p, hence the remaining occurrence cannot be evaluated anymore; this definition contains a fork in p. Since, in this case, the same destructive operation is applied to p, the fork can be removed using a where-clause:

$$f_3.p = \mathsf{cons}.(\mathsf{hd}.p).(\mathsf{cons}.(\mathsf{hd}.q).(\mathsf{tl}.q)) \;|[\, q = \mathsf{tl}.p \,]|.$$

To show that it is necessary to bound the number of transformations in the right-hand side of a function definition rather than the number of destructive operations, we consider:

$$f_4.p = (\mathsf{tl}.p, \mathsf{cons}.a.p).$$

In this case, evaluation of tl.p destroys p, hence also cons.$a.p$ is mutilated. The problem is that transformation cons.$a.p$ creates an extra reference to p, and therefore its use is also restricted.

The necessity of restricting transformations other than tl and cons as well, follows from the following refinement of f_4:

$$f_5.p = (\mathsf{tl}.q, \mathsf{cons}.a.r) \;|[\; (q, r) = (p, p) \;]|.$$

In this definition no stack-operations are applied to p. Nevertheless, evaluation of $f_5.p$ does not yield the desired result. The problem is that p is duplicated in the where-clause by means of two applications of the identity function. Another example of this phenomenon is

$$f_6.p = (\mathsf{tl}.q, \mathsf{cons}.a.p) \;|[\; q = p \;]|.$$

This definition also contains a fork in p.

Remark 4.2
To obtain programs without forks for the above functions, duplication can be programmed as follows:

$$\begin{array}{lcll} \mathsf{dup}.p & = & (\mathsf{empty}, \mathsf{empty}) & , \mathsf{isempty}.p \\ & [] & (\mathsf{cons}.(\mathsf{hd}.p).q, \mathsf{cons}.(\mathsf{hd}.p).r) & \\ & & |[\ (q,r) = \mathsf{dup}.(\mathsf{tl}.p)\ ]| & , \neg\mathsf{isempty}.p. \end{array}$$

This operation is a destructive transformation, hence its usage must be restricted as well.

Another approach is to define operation copy which refines the identity on $[T]$:

$$\begin{array}{lcll} \mathsf{copy}.p & = & \mathrm{nil} & , p = \mathrm{nil} \\ & [] & |[\mathrm{var}\ h{:}L; & \\ & & ;\ \mathrm{new}(h) & \\ & & ;\ h^\wedge := \langle p^\wedge.a, \mathsf{copy}.(p^\wedge.r)\rangle;\ \mathrm{return}(h) & \\ & &]| & , p \neq \mathrm{nil}. \end{array}$$

The use of copy need not be restricted because the value of p is merely inspected, not altered. In other words, although the type of this operation is $L \curvearrowright L$, copy should not be considered as a transformation, but as an inspection followed by a creation. Note that copy cannot be programmed nondestructively in terms of stack operations because tl is destructive.
□

Finally, we present two examples involving conditional expressions. As a rule, transformations should not be used in guards of conditional expressions, but guarded expressions may contain one transformation per variable each.

Example 4.3
The following program for list reversal is nondestructive when the nondestructive implementation of stacks is used:

$$\begin{array}{ll} \mathsf{rev}.x = & h.x.[\,] \\ & |[\ h.x.y \;\; = \;\; y \qquad\qquad\qquad\qquad\quad , \mathsf{isempty}.x \\ & \qquad\qquad\quad [] \;\; h.(\mathsf{tl}.x).(\mathsf{cons}.(\mathsf{hd}.x).y) \;\; , \neg\mathsf{isempty}.x \\ &]|. \end{array}$$

However, since both x and y are used in a linear fashion in the definition of h, this program also allows for the use of a destructive implementation of stacks. This yields a destructive implementation of list reversal, which is in fact an *in-situ* implementation, since the cell disposed by tl can be recycled by the operation new invoked by cons.
□

Example 4.4
Let $\bowtie$ denote the operator that merges two ascending lists of integers into a single ascending list. With this operator at our disposal we can write a version of mergesort that uses linear recursion only:

$$
\begin{array}{lll}
\mathit{Mergesort} = & g \circ f & \\
& \|[\ f.[\,] = \mathsf{empty} & \\
& \quad f.(a \vdash s) = \mathsf{snoc}.(f.s).[a] & \\
& \quad g.x = [\,] & ,\mathsf{isempty}.x \\
& \qquad [\!]\ \ (\ \mathsf{hd}.x & ,\mathsf{isempty}.y \\
& \qquad\quad [\!]\ \ g.(\mathsf{snoc}.(\mathsf{tl}.y).(\mathsf{hd}.x \bowtie \mathsf{hd}.y)) & ,\neg\mathsf{isempty}.y \\
& \qquad\quad)\ \|[\ y = \mathsf{tl}.x\ ]\| & ,\neg\mathsf{isempty}.x \\
&]\|. &
\end{array}
$$

To allow for the use of a destructive implementation of queues, variable y has been introduced to see to it that x is used in a linear way. Note that function f creates a queue and that g destructively inspects a queue.
□

The last example shows that our rule for linear usage is not adequate in general, since it does not restrict the use of inspection g, although it is destructive. A better rule for a definition $f.x = F$ seems to be that evaluation of F gives rise to at most one application of a transformation or a destructive inspection to x. This is however a little bit too restrictive, as we shall see in the next section.

4.7 Benevolent side-effects

Crucial for the efficiency of some data structures is that inspections "rearrange" their argument so that later operations take less time. These rearrangements are harmless in the sense that the abstract value represented by the argument remains the same. Such inspections are said to have *benevolent side-effects* [16].

As a somewhat contrived example of such an inspection, we consider the following version of the stack-operation hd:

$$\mathsf{hd}.p = \mathrm{return}(p^\wedge.a);\ \ |[\mathrm{var}\ h{:}L;\ \ \mathrm{new}(h);\ \ h^\wedge := p^\wedge;\ \ \mathrm{dispose}(p);\ \ p := h\ ]|.$$

Note that the value of p is altered by $\mathsf{hd}.p$, but that the value of $[\![p]\!]$ remains intact. A possible advantage of this version of hd is that by selection of an appropriate cell by $\mathrm{new}(h)$, the cells in use can be kept in a more contiguous part of memory. In Part II we will encounter more appealing examples of inspections with benevolent side-effects (e.g., operation member in Section 11.2.1 on splay trees).

To discuss the correctness of inspections with side-effects, we associate with each inspection $g \in C \to T$ a function $g' \in C \to T{\times}C$ such that $g'.c = (g.c, d)$, where d equals the value of c after evaluation of $g.c$. With the corresponding abstract inspection $f \in A \to T$ we associate function $f' \in A \to T{\times}A$ with $f'.a = (f.a, a)$. Then f is said to be data-refined by g under coupling $\simeq$ when f' is data-refined by g' under coupling $\simeq$. Hence, an inspection with benevolent side-effects is a mixture of an inspection and a transformation.

Since inspections with benevolent side-effects do not alter the abstract value of their argument, they can be treated as nondestructive inspections. In summary, we thus have the following restriction on the usage of a parameter that belongs to a destructive algebra.

> For a function definition of the form $f.x = F$, x is said to be used in a linear fashion when each evaluation of F gives rise to at most one application of a transformation or a destructive inspection to x, not counting inspections with benevolent side-effects.

As before, this rule refers to transformations and inspections that are either operations of the algebra or user-defined functions.

Chapter 5

Analysis of functional programs and algebras

As programs are designed in a modular way, we wish to analyze them in a modular way as well. In particular, we want to determine the complexity of a program from the complexities of the operations of the algebras used in the program. For instance, given the complexities of zero and suc, we want to determine the complexity of $\mathsf{suc}^n.\mathsf{zero}$ as a function of n. We call this a *compositional* way of analyzing programs, which—in essence—means that the complexity of a composition $g \circ f$, say, can be expressed in terms of the complexities of f and g.

Since it is often difficult—or even infeasible—to determine the exact complexity of a program, it is customary to work with approximations. Traditionally, worst-case bounds are used for this purpose, but this approach does not give satisfactory results in all circumstances: the worst-case complexity of $g \circ f$ may be considerably "better" than the sum of the worst-case complexities of f and g. In other words, worst-case complexity is not compositional. As explained in Chapter 3, we will therefore use amortized costs instead of actual costs to describe the efficiency of implementations of algebras.

To guarantee that the amortized cost of a program is the sum of the amortized costs of the operations used by it, the corresponding algebras should be used linearly—in the same way as destructive algebras should be used linearly to guarantee the correctness of a program. This connection between destructivity and amortization leads to interesting trade-offs, as will be shown at the end of this chapter.

5.1 Cost measures

As explained in Section 4.2, execution of a functional program f on an input x boils down to the reduction of expression $f.x$. Such a reduction requires time as well as space and it is the goal of an *analysis* to quantify the use of these resources. More concretely, the goal of an analysis of a program f is to express, as a function of x, the *cost* of evaluating $f.x$. Here, the cost depends on the choice of a *cost measure*, which is chosen beforehand. Since execution of functional programs amounts to the reduction of expressions, our cost measures will be mappings from expressions to the real numbers.

There are two—usually conflicting—demands that determine the suitability of a cost measure: it should be *realistic* for the intended purpose, and it should be *manageable*, i.e., not overly complicated. For instance, cost measure $\mathcal{N}$, defined by $\mathcal{N}(E)$= "the total number of unfoldings needed to evaluate E", is in general a realistic measure but often much too complicated. For a sorting program, a measure like "the number of comparisons needed to evaluate E" is probably more suitable; this measure counts the number of unfoldings of the predefined operation $<$, say.

Once a suitable cost measure has been defined, the program can be analyzed. We want to do so by analyzing the auxiliary functions, which constitute the program, one at a time. To support such a modular analysis, we introduce the "cost of a function".

Definition 5.1
A *cost measure* is a mapping from expressions to the real numbers. For a functional program f, the *cost of* f with respect to cost measure $\mathcal{T}$ is the mapping $\mathcal{T}[f]$ defined by

$$\mathcal{T}[f](E) = \mathcal{T}(f.E) - \mathcal{T}(E),$$

for $E \in \operatorname{dom} f$.
□

The important property of mapping $\mathcal{T}[\cdot]$ is that it satisfies the *composition rule*, which enables us to decompose the analysis of compositions like $g \circ f$.

Property 5.2 (composition rule for $\mathcal{T}[\cdot]$)
If $\mathcal{T}((g \circ f).E) = \mathcal{T}(g.(f.E))$, then

$$\mathcal{T}[g \circ f](E) = \mathcal{T}[f](E) + \mathcal{T}[g](f.E).$$

Proof

$$\begin{array}{ll}
 & \mathcal{T}[g \circ f](E) \\
= & \{ \text{ Definition 5.1 } \} \\
 & \mathcal{T}((g \circ f).E) - \mathcal{T}(E) \\
= & \{ \text{ above proviso } \} \\
 & \mathcal{T}(g.(f.E)) - \mathcal{T}(E) \\
= & \{ \text{ arithmetic } \} \\
 & \mathcal{T}(g.(f.E)) - \mathcal{T}(f.E) + \mathcal{T}(f.E) - \mathcal{T}(E) \\
= & \{ \text{ Definition 5.1 } \} \\
 & \mathcal{T}[g](f.E) + \mathcal{T}[f](E).
\end{array}$$

□

The proviso in this property is rather weak; it is satisfied by all cost measures that we use for our programs. This has to do with the fact that we use $\circ$ as an abbreviation mechanism only, and therefore we do not want that expressions $(g \circ f).E$ and $g.(f.E)$ are distinguished by our cost measures. Fancy cost measures that do not satisfy this proviso, such as "the length of expression E" or "the number of occurrences of $\circ$ in E", will not be used.

At first sight one might think that another property of $\mathcal{T}[f]$ is that $\mathcal{T}[f](E)$ depends on the *value* of E only, not on the *shape* of E, but this is not true in general. Take, for instance, cost measure $\mathcal{S}$ defined by $\mathcal{S}(E)$="the maximal amount of storage space in use during the evaluation of E". Since the maximal amount of storage space in use during the evaluation of $f.E$ is either in use before f is unfolded, in which case $\mathcal{S}(f.E) = \mathcal{S}(E)$, or only after f has been unfolded, in which case $\mathcal{S}(f.E) > \mathcal{S}(E)$, we observe that $\mathcal{S}[f](E)$ may be zero as well as positive, depending on the shape of E.

In the sequel, however, we will concentrate on cost measures that are related to the amount of time required for the evaluation of expressions. We call these *distributive cost measures.* The class of distributive cost measures will not be defined explicitly: we only postulate some *distribution rules* which are satisfied by such cost measures.

One of the most important distribution rules for a cost measure $\mathcal{T}$ is that for a function definition of the form $f.x = F$:

$$\mathcal{T}(f.E) = \mathcal{T}(E) + C_f(val(E)) + \mathcal{T}(F^x_{val(E)}),$$

where $val(E)$ denotes the value of E, and $C_f \in \text{dom}\, f \rightarrow \text{Real}$ describes the actual cost of unfolding f's definition for each possible value. Since eager evaluation of $f.E$ amounts to the evaluation of E, followed by the unfolding of f's definition, which in turn gives rise to the evaluation of F with x replaced by the value of E (cf. Section 4.2), we see that the cost of $f.E$ equals the sum of the costs of these three phases for distributive cost measures. Hence, the value of E is computed only once and *shared* by all occurrences of x in F. A consequence of this distribution rule is that

$$\mathcal{T}[f](E) = \mathcal{T}[f](val(E)),$$

since $val(val(E)) = val(E)$.

In the applications in Part II, all cost measures count unfoldings of (usually recursive) user-defined functions and/or unfoldings of predefined operations. Since we assume that our programs are executed according to the eager evaluation scheme (Section 4.2), these cost measures enjoy some nice distributivity properties. Below, we will present these properties for an extreme member of this class, viz. the previously introduced cost measure $\mathcal{N}$, which counts *all* unfoldings of predefined operations and user-defined functions.

Distribution rules for $\mathcal{N}$

1. The first rule is an instance of the above distribution rule for $\mathcal{T}(f.E)$, where f is defined by $f.x = F$:

$$\mathcal{N}(f.E) = \mathcal{N}(E) + 1 + \mathcal{N}(F^x_{val(E)}).$$

2. With respect to $\circ$, we have (cf. Property 5.2):

$$\mathcal{N}((g \circ f).E) = \mathcal{N}(g.(f.E)).$$

3. For conditional expressions the following rule is appropriate:

$$\mathcal{N}((E, B \,[]\, F, \neg B)) = \mathcal{N}(B) + \begin{cases} \mathcal{N}(E) & ,B \\ \mathcal{N}(F) & ,\neg B, \end{cases}$$

which expresses that in case of complementary guards, only one of these is evaluated, and that, subsequently, only the relevant expression is evaluated.

4. As for where-clauses, we consider expressions of the form $F|[x = E]|$. These are reduced in the same way as $f.E$ is reduced. Hence:

$$\mathcal{N}(F|[x = E]|) = \mathcal{N}(E) + 1 + \mathcal{N}(F^x_{val(E)}).$$

5. Another simple rule is that for a predefined binary operator $\oplus$, say, we have

$$\mathcal{N}(E \oplus F) = \mathcal{N}(E) + \mathcal{N}(F) + 1,$$

which reflects the fact that $\mathcal{N}$ counts unfoldings of predefined operations.

6. In connection with the use of patterns in the left-hand sides of function definitions, we consider a definition of the form

$$\begin{array}{lcl} f.[\,] & = & F \\ f.(a \vdash s) & = & G. \end{array}$$

For this definition we have the property that

$$\mathcal{N}(f.E) = \mathcal{N}(E)+1+1+ \begin{cases} \mathcal{N}(F) & ,E = [\,] \\ 1+1+\mathcal{N}(G^{a,s}_{val(hd.E),val(tl.E)}) & ,E \neq [\,]. \end{cases}$$

It may be interpreted as follows: first E is evaluated and it is determined whether E is empty or not, which requires one unfolding of $(=[\,])$. Subsequently f's definition is unfolded, which gives rise to evaluation of F *or* to evaluation of G with a and s replaced by the values of $hd.E$ and $tl.E$. The latter case requires one unfolding of hd and one of tl.

7. Finally, there is the trivial but useful property that evaluation of an expression terminates as soon as it has been reduced to its value, which is reflected by the rule

$$\mathcal{N}(val(E)) = 0.$$

These rules hold for all cost measures $\mathcal{T}$ that count unfoldings of user-defined and/or predefined functions, except that the "+1 terms" in the equations for $\mathcal{T}(f.E)$, $\mathcal{T}(E \oplus F)$, and $\mathcal{T}(F\|[x = E]\|)$ are absent for those unfoldings that are not counted by $\mathcal{T}$.

To facilitate the definition of distributive cost measures, we use *dots* on top of the =-signs in function definitions to mark the unfoldings that are counted by the measure. In case a function definition consists of several alternatives the dot marks *all* the alternatives. For example, the cost measure $\mathcal{T}$ defined by

$$\begin{array}{lcll} bubble.[\,] & = & [\,] & \\ bubble.[a] & = & [a] & \\ bubble.(a \vdash b \vdash s) & \doteq & a \vdash bubble.(b \vdash s) & , a \leq b \\ & [\!] & b \vdash a \vdash bubble.s & , a > b \end{array}$$

satisfies $\mathcal{T}(E)$="the number of unfoldings of the recursive alternative of *bubble* needed to evaluate E", and also $\mathcal{T}(E)$="the number of unfoldings of $\leq$ (or $>$) needed to evaluate E".

The distribution rules for $\mathcal{T}$ enable us to derive relations for $\mathcal{T}[f]$ that follow the structure of the definition of f. In case f's definition is recursive, this gives rise to recurrence relations for $\mathcal{T}[f]$. For example, using Definition 5.1, we have as recurrence relation for $\mathcal{T}[bubble]$:

$$\begin{array}{lcl} \mathcal{T}[bubble]([\,]) & = & 0 \\ \mathcal{T}[bubble]([a]) & = & 0 \\ \mathcal{T}[bubble](a \vdash b \vdash s) & = & 1 + \left\{ \begin{array}{ll} \mathcal{T}[bubble](b \vdash s) & , a \leq b \\ \mathcal{T}[bubble](s) & , a > b. \end{array} \right. \end{array}$$

Using the distribution rules, this recurrence relation may actually be derived, but this is too laborious and therefore omitted. Note that it suffices to investigate $\mathcal{T}[bubble](s)$ for all *values* s because $\mathcal{T}$ is distributive.

5.2 Worst-case analysis

It seems that we have already reached our goal now that we have introduced $\mathcal{T}[\cdot]$, since it satisfies some nice composition and distribution properties when $\mathcal{T}$ is distributive. Unfortunately, however, an explicit formula for $\mathcal{T}[f]$ cannot always be given, and in such a case we must content ourselves with approximations of $\mathcal{T}[f]$. A common method of approximating the cost of a function f is to use upper bounds for $\mathcal{T}[f](x)$ that depend on the "size" of x only. In these so-called *worst-case analyses*, the input values are partitioned into classes of equal-sized inputs, and the *worst-case complexity* of a function f is then determined for each possible size N. That is, with $\#x$ denoting the size of x,

$$(\mathrm{Max}\, x : x \in \mathrm{dom}\, f \;\wedge\; \#x = N : \mathcal{T}[f](x))$$

is determined as a function of N. For program *bubble*, for example, the worst case occurs when s is ascending; the solution of the recurrence relation then is $\mathcal{T}[\mathit{bubble}](s) = (\#s{-}1) \max 0$.

In many cases it is even too complicated to find an explicit formula for the worst-case complexity. Instead of an exact formula we then give an upper bound like $\#s$ for $\mathcal{T}[\mathit{bubble}](s)$, or we simply say that $\mathcal{T}[\mathit{bubble}](s)$ is $O(\#s)$ when we are only interested in the *asymptotic cost*. The problem with such approximations is that they are *not compositional* in the sense that a tight upper bound for the worst-case complexity of $\mathcal{T}[g \circ f]$ cannot be obtained from tight upper bounds for the worst-case complexities of $\mathcal{T}[f]$ and $\mathcal{T}[g]$. This is illustrated by the next example.

Example 5.3
We analyze a binary implementation of $(\,\mathrm{Nat} \mid 0, (+1)\,)$. The concrete data type is $[\{0,1\}]$ and the concrete operations are

$$\begin{array}{lcl}
\mathsf{zero} & \doteq & [\,] \\
\mathsf{suc}.[\,] & \doteq & [1] \\
\mathsf{suc}.(0 \vdash s) & \doteq & 1 \vdash s \\
\mathsf{suc}.(1 \vdash s) & \doteq & 0 \vdash \mathsf{suc}.s.
\end{array}$$

The cost measure defined by the dots is called $\mathcal{T}$, which could also be done in words by $\mathcal{T}(E)$="the number of unfoldings of zero and suc needed to evaluate E". Our goal is to derive a tight bound for $\mathcal{T}[\mathsf{suc}^n.\mathsf{zero}]$ as a function of n.

To this end we first analyze zero and suc in isolation. Clearly, $\mathcal{T}[\mathsf{zero}] = 1$, and for $\mathcal{T}[\mathsf{suc}]$ we have the following recurrence relation:

$$\begin{array}{lcl}
\mathcal{T}[\mathsf{suc}]([\,]) & = & 1 \\
\mathcal{T}[\mathsf{suc}](0 \vdash s) & = & 1 \\
\mathcal{T}[\mathsf{suc}](1 \vdash s) & = & 1 + \mathcal{T}[\mathsf{suc}](s).
\end{array}$$

A tight upper bound for $\mathcal{T}[\mathsf{suc}](s)$ in terms of $\#s$ thus is $1{+}\#s$.

Repeatedly applying the composition rule for $\mathcal{T}[\cdot]$ yields

$$\mathcal{T}[\mathsf{suc}^n.\mathsf{zero}] = \mathcal{T}[\mathsf{zero}] + (\Sigma\, i : 0 \le i < n : \mathcal{T}[\mathsf{suc}](\mathsf{suc}^i.\mathsf{zero})).$$

In order to use the upper bound for $\mathcal{T}[\mathsf{suc}]$, we observe that list $\mathsf{suc}^i.\mathsf{zero}$ has length at most $1{+}\log_2(i{+}1)$. Therefore $\mathcal{T}[\mathsf{suc}](\mathsf{suc}^i.\mathsf{zero})$ is at most $2{+}\log_2(i{+}1)$, and we obtain an $O(n \log n)$ bound for $\mathcal{T}[\mathsf{suc}^n.\mathsf{zero}]$.

However, $\mathcal{T}[\mathsf{suc}^n.\mathsf{zero}]$ is $O(n)$, as will be shown in Example 5.7; the fact that $\mathcal{T}[\mathsf{suc}](s) \le \#s$ (and that this bound is tight) cannot be used to prove this.
□

5.3 Amortized cost of functions

Suppose that our goal is to determine the worst-case complexity of f^n as a function of n. As demonstrated in Example 5.3, the worst-case complexity of f does not, in general, provide a clue to the worst-case complexity of f^n. Instead, we will use *amortized costs* for f and f^n, which are defined in terms of a *potential function.* In the functional setting, a potential function is—like a cost measure—a mapping on expressions. Unlike a cost measure, however, the potential of an expression depends on its value only, not on the way in which this value is expressed. This corresponds to the idea behind potential functions in Chapter 3: in the imperative setting, a potential function depends on the value of the state only, not on how the state has been reached.

Definition 5.4
A *potential function* Φ is a mapping from expressions to the real numbers satisfying

$$\Phi.E = \Phi.val(E).$$

The *amortized cost of* f with respect to cost measure $\mathcal{T}$ and potential Φ is the mapping $\mathcal{A}[f]$ defined by

$$\mathcal{A}[f](E) = \mathcal{T}[f](E) + \Phi.(f.E) - \Phi.E,$$

for $E \in \operatorname{dom} f$.
□

In an amortized analysis, the potential function is defined for the relevant values only, e.g., for the data type of an algebra. For other values, the definition of the potential is extended in a straightforward way. In connection with the use of tuples, for example, the potential of the empty tuple $\perp$ is usually defined equal to 0. For constant f, Definition 5.4 then yields

$$\mathcal{A}[f] = \mathcal{T}[f] + \Phi.f.$$

Also, in connection with the use of pairs, $\Phi.(E, F)$ is usually defined equal to the sum of $\Phi.E$ and $\Phi.F$, so that

$$\mathcal{A}[f]((E,F)) = \mathcal{T}[f]((E,F)) + \Phi.(f.E.F) - \Phi.E - \Phi.F$$

for functions f on pairs.

Since $\Phi.E = \Phi.val(E)$, $\mathcal{A}[\cdot]$ inherits a number of properties of $\mathcal{T}[\cdot]$. For instance, the composition rule is inherited.

Property 5.5 (composition rule for $\mathcal{A}[\cdot]$)
If

$$\mathcal{T}[g \circ f](E) = \mathcal{T}[f](E) + \mathcal{T}[g](f.E),$$

then also

$$\mathcal{A}[g \circ f](E) = \mathcal{A}[f](E) + \mathcal{A}[g](f.E).$$

Proof

$$\begin{array}{ll} & \mathcal{A}[g \circ f](E) \\ = & \{ \text{ Definition 5.4 } \} \\ & \mathcal{T}[g \circ f](E) + \Phi.((g \circ f).E) - \Phi.E \\ = & \{ \text{ above proviso and restriction on } \Phi \text{ in Definition 5.4 } \} \\ & \mathcal{T}[f](E) + \mathcal{T}[g](f.E) + \Phi.(g.(f.E)) - \Phi.E \\ = & \{ \text{ arithmetic } \} \\ & \mathcal{T}[g](f.E) + \Phi.(g.(f.E)) - \Phi.(f.E) + \mathcal{T}[f](E) + \Phi.(f.E) - \Phi.E \\ = & \{ \text{ Definition 5.4 } \} \\ & \mathcal{A}[g](f.E) + \mathcal{A}[f](E). \end{array}$$

□

Also “independence of the shape of E” is inherited by $\mathcal{A}[f](E)$.

Property 5.6
If $\mathcal{T}[f](E) = \mathcal{T}[f](val(E))$, then $\mathcal{A}[f](E) = \mathcal{A}[f](val(E))$ as well.
□

There is however a crucial difference between $\mathcal{A}[\cdot]$ and $\mathcal{T}[\cdot]$. To discuss this difference, we introduce cost measure $\mathcal{A}$:

$$\mathcal{A}(E) = \mathcal{T}(E) + \Phi.E.$$

Then the amortized cost of f w.r.t. $\mathcal{T}$ and Φ (cf. Definition 5.4) equals the cost of f w.r.t. $\mathcal{A}$ (cf. Definition 5.1). Since $\mathcal{A}$ inherits the common properties of $\mathcal{T}$ and Φ, Properties 5.5 and 5.6 are readily seen to be valid. However, distributivity of $\mathcal{T}$ does not imply distributivity of $\mathcal{A}$ because Φ is not distributive!

Now, since $\mathcal{T}$ is usually distributive, it is not difficult to obtain a relation for $\mathcal{T}[f]$ based on the structure of f's definition. But $\mathcal{A}$ is not distributive, and therefore a relation for $\mathcal{A}[f]$ cannot be obtained in general by substituting $\mathcal{A}$ for $\mathcal{T}$ in the relation for $\mathcal{T}[f]$. Consider, for instance, function *dup* defined by

$$dup.x \doteq (x, x).$$

Assuming that $\mathcal{T}$ is distributive and that Φ distributes over pairs, hence that $\mathcal{A}$ distributes over pairs (i.e., $\mathcal{A}((x,y)) = \mathcal{A}(x) + \mathcal{A}(y)$), we observe for value x:

$$\begin{array}{cl} & \mathcal{T}[dup](x) \\ = & \{ \text{ Definition 5.1 } \} \\ & \mathcal{T}(dup.x) - \mathcal{T}(x) \\ = & \{ \text{ definition } dup \} \\ & 1 + \mathcal{T}((x,x)) - \mathcal{T}(x) \\ = & \{ \mathcal{T} \text{ is distributive } \} \\ & 1 + \mathcal{T}(x) + \mathcal{T}(x) - \mathcal{T}(x) \\ = & \{ \mathcal{T}(x) = 0 \} \\ & 1, \end{array} \qquad \begin{array}{cl} & \mathcal{A}[dup](x) \\ = & \{ \text{ Definition 5.1 } \} \\ & \mathcal{A}(dup.x) - \mathcal{A}(x) \\ = & \{ \text{ definition } dup \} \\ & 1 + \mathcal{A}((x,x)) - \mathcal{A}(x) \\ = & \{ \mathcal{A} \text{ distributes over pairs } \} \\ & 1 + \mathcal{A}(x) + \mathcal{A}(x) - \mathcal{A}(x) \\ = & \{ \mathcal{T}(x) = 0, \text{ hence } \mathcal{A}(x) = \Phi.x \} \\ & 1 + \Phi.x. \end{array}$$

Hence, $\mathcal{T}[dup](x) = 1$ but $\mathcal{A}[dup](x) = 1 + \Phi.x$. Similarly, when x is tripled, the difference becomes $2\,\Phi.x$, and so on. On the other hand, for function $sink$, with $sink.x \doteq \perp$, we have that $\mathcal{T}[sink](x) = 1$ and $\mathcal{A}[sink](x) = 1 - \Phi.x$.

In general, for a definition of the form $f.x = F$, there will be no difference between the relations for $\mathcal{A}[f]$ and $\mathcal{T}[f]$ when x occurs exactly once in F (and $\mathcal{T}$ is distributive). This is reflected by the difference between

$$\mathcal{T}[f](E) = C_f(val(E)) + \mathcal{T}(F^x_{val(E)}),$$

and

$$\mathcal{A}[f](E) = C_f(val(E)) + \mathcal{A}(F^x_{val(E)}) - \Phi.E.$$

(Note that the equality for $\mathcal{T}[f](E)$ implies the equality for $\mathcal{A}[f](E)$.) To rewrite $\mathcal{T}(F^x_{val(E)})$ into a formula in terms of $\mathcal{T}[\cdot]$, as many terms $\mathcal{T}(val(E))$ can be used as necessary (since $\mathcal{T}(val(E)) = 0$), but to rewrite $\mathcal{A}(F^x_{val(E)})$ into a formula in terms of $\mathcal{A}[\cdot]$, only a single term $\Phi.E$ is available.

In Section 5.5 we will present a simple method to ensure that the relations for $\mathcal{T}[\cdot]$ are inherited by $\mathcal{A}[\cdot]$.

5.4 Amortized analysis

Consider a function f for which $\operatorname{rng} f \subseteq \operatorname{dom} f$. Suppose that the worst-case complexity of f^n (as a function of n) cannot be determined from the worst-case complexity of f. Then we perform an *amortized analysis*, which typically proceeds as follows. For the sake of convenience we assume that $\#(f.x) = \#x$.

Having defined cost measure $\mathcal{T}$, our aim is to bound $\mathcal{T}[f^n](x)$ in terms of $\#x$ and n. To this end we repeatedly apply the composition rule for $\mathcal{A}[\cdot]$ (Property 5.5) to $\mathcal{A}[f^n]$, where

$$\mathcal{A}[f^n](x) = \mathcal{T}[f^n](x) + \Phi.(f^n.x) - \Phi.x.$$

This yields the following equation:

(1) $\mathcal{T}[f^n](x) = (\Sigma\, i : 0 \le i < n : \mathcal{A}[f](f^i.x)) + \Phi.x - \Phi.(f^n.x)$,

from which we obtain a tight bound for $\mathcal{T}[f^n](x)$ by choosing a suitable potential Φ. But, just as it is difficult to find an explicit formula for $\mathcal{T}[f](x)$, it is, in general, difficult to find an explicit formula for $\mathcal{A}[f](x)$. Therefore, we use the *worst-case* approximation of $\mathcal{A}[f]$ by considering the following bound for $\mathcal{T}[f^n]$ derived from (1), using that $\#(f.x) = \#x$:

$$\mathcal{T}[f^n](x) \le n * (\mathrm{Max}\, y : \#y = \#x : \mathcal{A}[f](y)) + \Phi.x - \Phi.(f^n.x).$$

The object is now to choose Φ such that $(\mathrm{Max}\, y : \#y{=}\#x : \mathcal{A}[f](y))$ is minimized, thereby keeping in mind that $\Phi.x - \Phi.(f^n.x)$ may not be too large as well.

As for the latter requirement on Φ, it is often convenient to choose a *nonnegative* function for Φ. The above upper bound then simplifies to

$$\mathcal{T}[f^n](x) \le n * (\mathrm{Max}\, y : \#y = \#x : \mathcal{A}[f](y)) + \Phi.x.$$

By keeping $\Phi.x$ small with respect to $\#x$, a tight upper bound for $\mathcal{T}[f^n](x)$ in terms of $\#x$ is obtained.

This above method of amortized analysis is applied in the next example to improve the analysis of $\mathsf{suc}^n.\mathsf{zero}$ in Example 5.3.

Example 5.7 (see Example 5.3)
The amortized costs for zero and suc are

$$\begin{array}{lcl} \mathcal{A}[\mathsf{zero}] & = & \mathcal{T}[\mathsf{zero}] + \Phi.\mathsf{zero} \\ \mathcal{A}[\mathsf{suc}](s) & = & \mathcal{T}[\mathsf{suc}](s) + \Phi.(\mathsf{suc}.s) - \Phi.s, \end{array}$$

where $\Phi \in [\{0,1\}] {\rightarrow} \mathrm{Real}$ is the potential function. Since

$$\mathcal{A}[\mathsf{suc}^n.\mathsf{zero}] = \mathcal{T}[\mathsf{suc}^n.\mathsf{zero}] + \Phi.(\mathsf{suc}^n.\mathsf{zero}),$$

the composition rule for $\mathcal{A}[\cdot]$ yields

$$\mathcal{T}[\mathsf{suc}^n.\mathsf{zero}] = \mathcal{A}[\mathsf{zero}] + (\Sigma\, i : 0{\le}i{<}n : \mathcal{A}[\mathsf{suc}](\mathsf{suc}^i.\mathsf{zero})) - \Phi.(\mathsf{suc}^n.\mathsf{zero}).$$

This equality yields an $O(n)$ bound for $\mathcal{T}[\mathsf{suc}^n.\mathsf{zero}]$ provided Φ is nonnegative, and $\mathcal{A}[\mathsf{zero}]$ and $\mathcal{A}[\mathsf{suc}](s)$ are both constant.

By definition, we have that $\mathcal{A}[\mathsf{zero}] = 1 + \Phi.[\,]$, hence constant. To keep $\mathcal{A}[\mathsf{zero}]$ small and Φ nonnegative, we take $\Phi.[\,] = 0$. Then $\mathcal{A}[\mathsf{zero}] = 1$.

Similarly, we have that $\mathcal{A}[\mathsf{suc}]([\,])$ is $O(1)$, independent of the definition of Φ. To *derive* a suitable definition for Φ, we first observe for the second alternative of suc:

$$\begin{array}{cl}
 & \mathcal{A}[\mathsf{suc}](0 \vdash s) \\
= & \{ \text{ definition of } \mathcal{A}[\mathsf{suc}] \ \} \\
 & \mathcal{T}[\mathsf{suc}](0 \vdash s) + \Phi.(\mathsf{suc}.(0 \vdash s)) - \Phi.(0 \vdash s) \\
= & \{ \text{ recurrence relation for } \mathcal{T}[\mathsf{suc}], \text{ definition of } \mathsf{suc} \ \} \\
 & 1 + \Phi.(1 \vdash s) - \Phi.(0 \vdash s) \\
= & \{ \text{ simplify by choosing } \Phi.(b \vdash s) = \varphi.b + \Phi.s \ \} \\
 & 1 + \varphi.1 - \varphi.0.
\end{array}$$

By the introduction of φ, $\mathcal{A}[\mathsf{suc}](0 \vdash s)$ becomes independent of s, hence $O(1)$. To obtain a suitable definition for φ we proceed with the last alternative of suc:

$$\begin{array}{cl}
 & \mathcal{A}[\mathsf{suc}](1 \vdash s) \\
= & \{ \text{ definition of } \mathcal{A}[\mathsf{suc}] \ \} \\
 & \mathcal{T}[\mathsf{suc}](1 \vdash s) + \Phi.(\mathsf{suc}.(1 \vdash s)) - \Phi.(1 \vdash s) \\
= & \{ \text{ recurrence relation for } \mathcal{T}[\mathsf{suc}], \text{ definition of } \mathsf{suc} \ \} \\
 & 1 + \mathcal{T}[\mathsf{suc}](s) + \Phi.(0 \vdash \mathsf{suc}.s) - \Phi.(1 \vdash s) \\
= & \{ \text{ definition of } \mathcal{A}[\mathsf{suc}] \ \} \\
 & \mathcal{A}[\mathsf{suc}](s) + 1 + \Phi.(0 \vdash \mathsf{suc}.s) - \Phi.(\mathsf{suc}.s) - (\Phi.(1 \vdash s) - \Phi.s) \\
= & \{ \ \Phi.(b \vdash s) = \varphi.b + \Phi.s, \text{ see previous derivation } \} \\
 & \mathcal{A}[\mathsf{suc}](s) + 1 + \varphi.0 - \varphi.1.
\end{array}$$

The term $1 + \varphi.0 - \varphi.1$ may be interpreted as the amortized cost of unfolding $\mathsf{suc}.(1 \vdash s)$.

To ensure that $\mathcal{A}[\mathsf{suc}]$ is $O(1)$, φ should be chosen such that $1+\varphi.0-\varphi.1 \leq 0$. To obtain a nonnegative and small Φ, we take $\varphi.0 = 0$ and $\varphi.1 = 1$ (hence, $1 + \varphi.0 - \varphi.1 = 0$). As potential we thus obtain:

$$\begin{array}{lcl}
\Phi.[\,] & = & 0 \\
\Phi.(b \vdash s) & = & b + \Phi.s,
\end{array}$$

hence $\Phi.s$ equals the number of 1's in s. From this definition for Φ, it follows by induction that $\mathcal{A}[\mathsf{suc}](s) = 2$. The above equation for $\mathcal{T}[\mathsf{suc}^n.\mathsf{zero}]$ then yields

$$\mathcal{T}[\mathsf{suc}^n.\mathsf{zero}] = 1 + 2n - \Phi.(\mathsf{suc}^n.\mathsf{zero}),$$

from which we conclude that $\mathcal{T}[\mathsf{suc}^n.\mathsf{zero}]$ is $O(n)$, since Φ is nonnegative. We remark that the upper bound $1+2n$ is as good as tight ($\mathcal{T}[\mathsf{suc}^n.\mathsf{zero}] = 2n$ when n is a power of two).
□

In this example, the potential has been derived in a calculational way. In particular, the fact that $\varphi.b = b$ is a suitable definition directly follows from the derivation. The systematic derivation of potential functions is an important topic in the case-studies of Part II.

5.5 Amortization and linearity

At this point it should be clear that worst-case *amortized* complexities should be used instead of worst-case *actual* complexities as a general way to describe the efficiency of implementations of algebras. But, as explained at the end of Section 5.3, a problem with the use of amortized costs is that for a function definition of the form $f.c = F$, say, the relations for $\mathcal{T}[f]$ are not necessarily inherited by $\mathcal{A}[f]$. Recall, for example, that $\mathcal{A}[dup](c) = 1 + \Phi.c$ whereas $\mathcal{T}[dup](c) = 1$ for $dup.c \doteq (c, c)$. To achieve that the relations for $\mathcal{T}[f]$ are indeed inherited by $\mathcal{A}[f]$ some restrictions must be imposed on expression F. Below we will describe these restrictions under the assumption that $\mathcal{A}[\cdot]$ is defined according to the following scheme.

Let C be a (concrete) monoalgebra with data type C. Let $\mathcal{T}$ denote a cost measure for the programs of C, and let Φ be a potential function defined on C. In accordance with Definition 5.4, the amortized costs of creations and transformations are defined as follows.

(a) For a creation g of type $T \to C$:

$$\mathcal{A}[g](x) = \mathcal{T}[g](x) + \Phi.(g.x).$$

(b) For a transformation g of type $C \times C \curvearrowright C$:

$$\mathcal{A}[g](c, d) = \mathcal{T}[g](c, d) + \Phi.(g.c.d) - \Phi.c - \Phi.d.$$

To obtain (a) as instantiation of Definition 5.4, we take $\Phi.x = 0$ for $x \in T$. Doing the same for inspections, however, yields a not so useful definition. Instead, we distinguish two types of inspections.

(c) For an inspection g of type $C \to T$:

(i) $\mathcal{A}[g](c) = \mathcal{T}[g](c)$, or
(ii) $\mathcal{A}[g](c) = \mathcal{T}[g](c) - \Phi.c$.

Here, (ii) can be obtained as instantiation of Definition 5.4, again with $\Phi.x = 0$ for $x \in T$. In most cases, however, (i) is the appropriate definition for $\mathcal{A}[g]$: defining $\mathcal{A}[g](c) = \mathcal{T}[g](c)$ means that the costs of inspections are *not* amortized.

If $\mathcal{A}[\cdot]$ is defined according to the above scheme, the restriction on $f.c = F$ is that each evaluation of F should give rise to at most one application to c of a transformation or an inspection of type (ii). In that case, $\mathcal{A}[f]$ inherits the relations for $\mathcal{T}[f]$. Note that the use of inspections of type (i) is not restricted by this rule. This is essential because inspections are typically used in combination with a transformation, and this would be impossible when the potential played a role in the amortized costs of both.

Inspections are allowed to have benevolent side-effects. In that case the potential of c may be altered by the evaluation of $g.c$, and this change in potential may be necessary to amortize the cost of g. To analyze such an inspection, we consider function $g' \in C \to T{\times}C$ with $g'.c = (g.c, d)$, where d equals the value of c after evaluation of $g.c$ (cf. Section 4.7). The amortized costs of g' are defined as follows:

$$\mathcal{A}[g'](c) = \mathcal{T}[g](c) + \Phi.d - \Phi.c.$$

In Part II we will see some examples of inspections with benevolent side-effects that are efficient in the amortized sense only (e.g., operation member in Section 11.2.1 on splay trees). In the above rule, such an inspection may be treated as an inspection of type (i).

Calling an inspection of type (i) a nondestructive inspection and an inspection of type (ii) a destructive one, the restriction on $f.c = F$ coincides with the notion of linearity defined at the end of Chapter 4. Therefore we shall say that an algebra which is efficient in the amortized sense only must be restricted to *linear usage* in order to ensure that the amortized costs add up in the same way as the actual costs. An important observation is now that an algebra which is either destructive or efficient in the amortized sense only might as well be both. This leads to interesting trade-offs, as we will show in the next section.

5.6 Purely-functional deques

In Chapter 7 of his Ph.D. thesis [17] Hoogerwoord derives an efficient implementation for a symmetric set of list operations. The idea behind his design is based on an efficient implementation of queues in LISP (see e.g. [13, pp. 250–251]). In this section, we first cast this design in our style, after which we go on to experiment with the amortized analysis of more advanced operations programmed in terms of the symmetric list operations. In this way we gain some experience with the computation of the amortized costs of composite functions from the amortized costs of the constituent parts.

5.6.1 Specification and implementation

Using the algebra of stacks

$$\mathsf{S} = (\,[T] \mid [\,], (=[\,]), \vdash, hd, tl\,),$$

whose operations all have $O(1)$ actual cost, we present an implementation of the algebra of *deques* ("double-ended queues")

$$\mathsf{DQ} = (\,[T] \mid [\,], (=[\,]), \vdash, hd, tl, \dashv, lt, ft, rev\,),$$

with signature

$$\mathsf{S2} = (\,\mathrm{X} \mid \mathsf{empty}, \mathsf{isempty}, \mathsf{cons}, \mathsf{hd}, \mathsf{tl}, \mathsf{snoc}, \mathsf{lt}, \mathsf{ft}, \mathsf{rev}\,),$$

such that all operations of S2 have $O(1)$ amortized cost. Apart from the symmetrical counterparts of the last three operations of S, we have included operation rev, because any program for rev in terms of the other operations of DQ is bound to be linear, while it is a simple $O(1)$ operation in the representation used for $[T]$ below. Hence, rev cannot be programmed without loss of efficiency in terms of the other operations of DQ.

The definition of S2 is as follows (see [17] for a solid derivation). Set X is a subset of $[T]\times[T]$ and S2 refines DQ under abstraction $[\![\cdot]\!]$ with $[\![\langle s,t\rangle]\!] = s +\!\!\!+\, rev.t$. To allow for an efficient implementation of hd and lt, set X is defined as follows:

$$\mathrm{X} = \{\langle s,t\rangle \mid (s=[\,] \;\Rightarrow\; \#t \le 1) \;\wedge\; (t=[\,] \;\Rightarrow\; \#s\le 1)\}.$$

Using the operations of S, the operations of S2 are programmed as follows:

$$\begin{array}{lcll}
\mathsf{empty} & = & \langle [\,],[\,]\rangle & \\[1ex]
\mathsf{isempty}.\langle s,t\rangle & = & s=[\,] \;\wedge\; t=[\,] & \\[1ex]
\mathsf{cons}.a.\langle s,t\rangle & = & \langle a\vdash s, t\rangle & ,t\neq[\,] \\
 & [\!] & \langle [a], s\rangle & ,t=[\,] \\[1ex]
\mathsf{hd}.\langle s,t\rangle & = & hd.s & ,s\neq[\,] \\
 & [\!] & hd.t & ,s=[\,] \\[1ex]
\mathsf{tl}.\langle s,t\rangle & = & \langle [\,],[\,]\rangle & ,\#s=0 \\
 & [\!] & \langle rev.(t{\downarrow}k), t{\uparrow}k\rangle \quad |[\; k=\#t \,\mathrm{div}\, 2\;]| & ,\#s=1 \\
 & [\!] & \langle tl.s, t\rangle & ,\#s\ge 2 \\[1ex]
\mathsf{snoc}.\langle s,t\rangle.a & = & \langle s, a\vdash t\rangle & ,s\neq[\,] \\
 & [\!] & \langle t, [a]\rangle & ,s=[\,] \\[1ex]
\mathsf{lt}.\langle s,t\rangle & = & hd.t & ,t\neq[\,] \\
 & [\!] & hd.s & ,t=[\,] \\[1ex]
\mathsf{ft}.\langle s,t\rangle & = & \langle [\,],[\,]\rangle & ,\#t=0 \\
 & [\!] & \langle s{\uparrow}k, rev.(s{\downarrow}k)\rangle \quad |[\; k=\#s \,\mathrm{div}\, 2\;]| & ,\#t=1 \\
 & [\!] & \langle s, tl.t\rangle & ,\#t\ge 2 \\[1ex]
\mathsf{rev}.\langle s,t\rangle & = & \langle t,s\rangle\;. &
\end{array}$$

In the programs for tl and ft some list operations are used which are not provided by algebra S. We assume that these operations are implemented in terms of the operations of S such that $rev.s$ and $\#s$ take $O(\#s)$ time, and, $s{\uparrow}n$ and $s{\downarrow}n$ take $O(n)$ time—as usual. Furthermore, an expression like $\#s{\ge}2$ is just short for $s{\neq}[\,]$ cand $tl.s{\neq}[\,]$, so it takes only $O(1)$ time to evaluate it.

Now, note that all operations are $O(1)$, except that evaluation of $\mathsf{tl}.\langle s,t\rangle$ takes $O(\#t)$ time in case $\#s{=}1$ (and similarly for ft). Therefore we count $1{+}\#t$ units for such an unfolding of tl, $1{+}\#s$ time units for the corresponding unfolding of ft, and in all other cases we just count one time unit per unfolding of an operation of $\mathsf{S2}$. This defines cost measure $\mathcal{T}$. The potential function is defined as $\Phi.\langle s,t\rangle = |\#s - \#t|$, and the amortized costs of the operations are all $O(1)$, with (cf. scheme in Section 5.5):

$$
\begin{array}{lcl}
\mathcal{A}[\mathsf{empty}] & = & \mathcal{T}[\mathsf{empty}] + \Phi.\mathsf{empty} \\
\mathcal{A}[\mathsf{isempty}](x) & = & \mathcal{T}[\mathsf{isempty}](x) \\
\mathcal{A}[\mathsf{cons}](a,x) & = & \mathcal{T}[\mathsf{cons}](a,x) + \Phi.(\mathsf{cons}.a.x) - \Phi.x \\
\mathcal{A}[\mathsf{hd}](x) & = & \mathcal{T}[\mathsf{hd}](x) \\
\mathcal{A}[\mathsf{tl}](x) & = & \mathcal{T}[\mathsf{tl}](x) + \Phi.(\mathsf{tl}.x) - \Phi.x \\
\mathcal{A}[\mathsf{snoc}](x,a) & = & \mathcal{T}[\mathsf{snoc}](x,a) + \Phi.(\mathsf{snoc}.x.a) - \Phi.x \\
\mathcal{A}[\mathsf{lt}](x) & = & \mathcal{T}[\mathsf{lt}](x) \\
\mathcal{A}[\mathsf{ft}](x) & = & \mathcal{T}[\mathsf{ft}](x) + \Phi.(\mathsf{ft}.x) - \Phi.x \\
\mathcal{A}[\mathsf{rev}](x) & = & \mathcal{T}[\mathsf{rev}](x) + \Phi.(\mathsf{rev}.x) - \Phi.x.
\end{array}
$$

In the analyses in the next section, we shall use that Φ satisfies $0{\le}\Phi.x{\le}\#[\![x]\!]$, for all $x{\in}X$.

To give an idea of how the correctness of the above results can be proved, we end this section with a treatment of case $\#s = 1$ for $\mathsf{tl}.\langle s,t\rangle$. The correctness of the program for this case follows from $\langle rev.(t{\downarrow}k), t{\uparrow}k\rangle \in X$ (since $k = \#t \operatorname{div} 2$), and

$$
\begin{array}{cl}
 & [\![\mathsf{tl}.\langle s,t\rangle]\!] \\
= & \quad \{ \text{ definition of } \mathsf{tl} \ \} \\
 & [\![\langle rev.(t{\downarrow}k), t{\uparrow}k\rangle]\!] \\
= & \quad \{ \text{ definition of } [\![\cdot]\!] \ \} \\
 & rev.(t{\downarrow}k) \mathbin{+\!\!+} rev.(t{\uparrow}k) \\
= & \quad \{ \text{ property of } rev;\ t{\uparrow}k \mathbin{+\!\!+} t{\downarrow}k = t \ \} \\
 & rev.t \\
= & \quad \{ \ \#s = 1 \ \} \\
 & tl.(s \mathbin{+\!\!+} rev.t) \\
= & \quad \{ \text{ definition of } [\![\cdot]\!] \ \} \\
 & tl.[\![\langle s,t\rangle]\!].
\end{array}
$$

The $O(1)$ bound for the amortized cost follows from (using $\mathcal{T}[\mathsf{tl}](\langle s,t\rangle) = 1{+}\#t$):

$$
\begin{array}{cl}
 & \mathcal{A}[\mathsf{tl}](\langle s,t\rangle) \\
= & \quad \{ \text{ definition of } \mathcal{A}[\mathsf{tl}] \ \} \\
 & \mathcal{T}[\mathsf{tl}](\langle s,t\rangle) + \Phi.(\mathsf{tl}.\langle s,t\rangle) - \Phi.\langle s,t\rangle
\end{array}
$$

$$
\begin{array}{ll}
= & \{ \text{ definitions of } \mathcal{T} \text{ and } \mathsf{tl} \ \} \\
 & 1 + \#t + |(\#t - k) - k| - |1 - \#t| \\
\leq & \{ \text{ arithmetic } (k = \#t \operatorname{div} 2) \ \} \\
 & 1 + \#t + \#t \bmod 2 + 1 - \#t \\
\leq & \{ \text{ arithmetic } \} \\
 & 3.
\end{array}
$$

5.6.2 Some applications and their analyses

In this section we perform a series of amortized analyses of more complicated list operations. Each example emphasizes a particular issue of amortized analysis. Given the choice for X in the previous section, all these operations are implemented without loss of efficiency, so there is no need to add any of these to the algebra of deques.

Catenation — *a simple amortized analysis*

For $+\!\!+$ we have as obvious refinement:

$$
\begin{array}{lcll}
\mathsf{cat}.x.y & = & y & ,\ \mathsf{isempty}.x \\
 & [] & \mathsf{cat}.(\mathsf{ft}.x).(\mathsf{cons}.(\mathsf{lt}.x).y) & ,\neg\mathsf{isempty}.x.
\end{array}
$$

Since the costs of the operations of S2 dominate, we charge the cost of each unfolding of cat to the applications of these operations. Thus, we leave the definition of $\mathcal{T}$ unchanged and define

$$\mathcal{A}[\mathsf{cat}](x,y) = \mathcal{T}[\mathsf{cat}](x,y) + \Phi.(\mathsf{cat}.x.y) - \Phi.x - \Phi.y,$$

since $+\!\!+$ is a transformation. Then the definition of cat is linear, so we have as recurrence relation for $\mathcal{A}[\mathsf{cat}]$:

$$
\mathcal{A}[\mathsf{cat}](x,y) = \begin{cases}
\mathcal{A}[\mathsf{isempty}](x) & ,\ [\![x]\!] = [\,] \\
\mathcal{A}[\mathsf{cat}](\mathsf{ft}.x, \mathsf{cons}.(\mathsf{lt}.x).y) + \mathcal{A}[\mathsf{ft}](x) + & \\
\mathcal{A}[\mathsf{cons}](\mathsf{lt}.x, y) + \mathcal{A}[\mathsf{lt}](x) + \mathcal{A}[\mathsf{isempty}](x) & ,\ [\![x]\!] \neq [\,],
\end{cases}
$$

from which we infer that $\mathcal{A}[\mathsf{cat}](x,y)$ is $O(\#[\![x]\!])$.

Of course, one can also design a program for $\mathsf{cat}.x.y$ with cost $O(\#[\![y]\!])$. By choosing the cheapest program, we obtain an implementation of cat for which $\mathsf{cat}.x.y$ takes $O(\#[\![x]\!] \min \#[\![y]\!])$ time, provided that we see to it that $\#[\![x]\!]$ and $\#[\![y]\!]$ can be determined in $O(1)$ time.

Length — *introducing an auxiliary definition*

Operation $\#$ may be refined by the following function:

$$
\begin{array}{lcll}
\mathsf{len}.x & = & 0 & ,\ \mathsf{isempty}.x \\
 & [] & 1 + \mathsf{len}.(\mathsf{tl}.x) & ,\neg\mathsf{isempty}.x.
\end{array}
$$

Since $\#$ is an inspection operation, we define:

$$\mathcal{A}[\mathsf{len}](x) = \mathcal{T}[\mathsf{len}](x).$$

However, in order to analyze len, we introduce the following auxiliary definition:

$$\mathcal{A}'[\mathsf{len}](x) = \mathcal{T}[\mathsf{len}](x) - \Phi.x,$$

for which we have the following recurrence relation:

$$\mathcal{A}'[\mathsf{len}](x) = \begin{cases} \mathcal{A}[\mathsf{isempty}](x) - \Phi.x & , [\![x]\!] = [\,] \\ \mathcal{A}'[\mathsf{len}](\mathsf{tl}.x) + \mathcal{A}[\mathsf{tl}](x) + \mathcal{A}[\mathsf{isempty}](x) & , [\![x]\!] \neq [\,]. \end{cases}$$

Since $\Phi.x \geq 0$, it follows that $\mathcal{A}'[\mathsf{len}](x)$ is $O(\#[\![x]\!])$, and, since $\Phi.x \leq \#[\![x]\!]$, we have that $\mathcal{A}[\mathsf{len}](x)$ is $O(\#[\![x]\!])$ as well.

From the above analysis we conclude that it is sometimes necessary to introduce a different definition for the amortized cost of an operation in order to perform the analysis of its program. In this case we need the term $-\Phi.x$ in the definition of $\mathcal{A}'[\mathsf{len}](x)$ to amortize the cost of the applications of tl in the recursive alternative of len. Subsequently, the bounds for Φ enabled us to derive a bound for $\mathcal{A}[\mathsf{len}](x)$.

Take and drop — *extending the definition of $\mathcal{T}$*

For $\uparrow$ ("take") and $\downarrow$ ("drop") we have as refinements

$$\begin{array}{lcl} \mathsf{take}.x.0 & = & \mathsf{empty} \\ \mathsf{take}.x.(n{+}1) & = & \mathsf{cons}.(\mathsf{hd}.x).(\mathsf{take}.(\mathsf{tl}.x).n), \end{array}$$

and

$$\begin{array}{lcl} \mathsf{drop}.x.0 & = & x \\ \mathsf{drop}.x.(n{+}1) & = & \mathsf{drop}.(\mathsf{tl}.x).n. \end{array}$$

With $\mathcal{T}$ as defined so far we have $\mathcal{T}[\mathsf{drop}](x,0) = 0$, which is evidently not realistic. Therefore, we extend the definition of $\mathcal{T}$ as follows: we count one unit per unfolding of the nonrecursive alternative of drop. The amortized costs of these transformations are

$$\begin{array}{lcl} \mathcal{A}[\mathsf{take}](x,n) & = & \mathcal{T}[\mathsf{take}](x,n) + \Phi.(\mathsf{take}.x.n) - \Phi.x \\ \mathcal{A}[\mathsf{drop}](x,n) & = & \mathcal{T}[\mathsf{drop}](x,n) + \Phi.(\mathsf{drop}.x.n) - \Phi.x. \end{array}$$

A straightforward analysis gives $O(n)$ bounds for the amortized costs of $\mathsf{take}.x.n$ and $\mathsf{drop}.x.n$.

Element selection — *less efficient than expected?*

We have $\mathsf{elt}.x.n$ as refinement for $[\![x]\!].n$, where

$$\begin{array}{lcl} \mathsf{elt}.x.0 & = & \mathsf{hd}.x \\ \mathsf{elt}.x.(n{+}1) & = & \mathsf{elt}.(\mathsf{tl}.x).n. \end{array}$$

Operation elt is an inspection, hence

$$\mathcal{A}[\mathsf{elt}](x,n) = \mathcal{T}[\mathsf{elt}](x,n).$$

In order to amortize the cost of the applications of tl, we introduce the following auxiliary definition:

$$\mathcal{A}'[\mathsf{elt}](x,n) = \mathcal{T}[\mathsf{elt}](x,n) - \Phi.x,$$

for which we have as recurrence relation:

$$\begin{array}{lcl} \mathcal{A}'[\mathsf{elt}](x,0) & = & \mathcal{A}[\mathsf{hd}](x) - \Phi.x \\ \mathcal{A}'[\mathsf{elt}](x,n{+}1) & = & \mathcal{A}'[\mathsf{elt}](\mathsf{tl}.x,n) + \mathcal{A}[\mathsf{tl}](x). \end{array}$$

Since $\Phi.x \geq 0$, $\mathcal{A}'[\mathsf{elt}](x,n)$ is $O(n)$, but for $\mathcal{A}[\mathsf{elt}](x,n)$ we can only conclude that it is $O(\#[\![x]\!])$, using that $\Phi.x \leq \#[\![x]\!]$. Now, given the choice for X in the previous section, this bound for $\mathcal{A}[\mathsf{elt}](x,n)$ is tight; the worst case occurs when the second element of $[\![\langle s,t\rangle]\!]$ is selected and $\#s = 1$, or when the last but one element of $[\![\langle s,t\rangle]\!]$ is selected and $\#t = 1$. So, element selection for deques is not as efficient as for stacks.

Splitting lists — *an avoidable fork*

As a kind of inverse of cat we consider operation split, which splits a list into a prefix of specified length and the remaining suffix:

$$\mathsf{split}.x.n = \langle \mathsf{take}.x.n, \mathsf{drop}.x.n \rangle.$$

Since split is a transformation with two outputs, we have

$$\mathcal{A}[\mathsf{split}](x,n) = \mathcal{T}[\mathsf{split}](x,n) + \Phi.(\mathsf{split}.x.n.0) + \Phi.(\mathsf{split}.x.n.1) - \Phi.x.$$

Now recall that the definitions of $\mathcal{A}[\mathsf{take}](x,n)$ and $\mathcal{A}[\mathsf{drop}](x,n)$ both contain term $-\Phi.x$, and observe that this term occurs only once in the definition of $\mathcal{A}[\mathsf{split}](x,n)$. This gives rise to a fork in x:

$$\mathcal{A}[\mathsf{split}](x,n) = \mathcal{A}[\mathsf{take}](x,n) + \mathcal{A}[\mathsf{drop}](x,n) + \Phi.x.$$

Since $\Phi.x{\leq}\#[\![x]\!]$, the best we can conclude from this relation is that $\mathcal{A}[\mathsf{split}](x,n)$ is $O(\#[\![x]\!])$, using that $\mathcal{A}[\mathsf{take}](x,n)$ and $\mathcal{A}[\mathsf{drop}](x,n)$ are $O(n)$. However, by tupling the programs for take and drop, we obtain an algorithmic refinement of split for which $\mathcal{A}[\mathsf{split}](x,n)$ can be shown to be $O(n)$:

$$\begin{array}{lcl} \mathsf{split}.x.0 & = & \langle \mathsf{empty}, x\rangle \\ \mathsf{split}.x.(n{+}1) & = & \langle \mathsf{cons}.(\mathsf{hd}.x).y, z\rangle \quad \|[\ \langle y,z\rangle = \mathsf{split}.(\mathsf{tl}.x).n\ ]\|. \end{array}$$

As in the previous program, x occurs twice in the right-hand side of the second alternative, but now only one transformation (viz. tl) is applied to it. Therefore,

$$\begin{array}{lcl} \mathcal{A}[\mathsf{split}](x,0) & = & \mathcal{A}[\mathsf{empty}] \\ \mathcal{A}[\mathsf{split}](x,n{+}1) & = & \mathcal{A}[\mathsf{split}](\mathsf{tl}.x,n) + \mathcal{A}[\mathsf{hd}](x) + \mathcal{A}[\mathsf{tl}](x) \\ & & +\mathcal{A}[\mathsf{cons}](\mathsf{hd}.x, \mathsf{split}.(\mathsf{tl}.x).n.0), \end{array}$$

which yields an $O(n)$ bound for the amortized costs of this program for split.

From this example we draw the important conclusion that it may be necessary to consider a more detailed refinement of a program in order to avoid forks. In this case, we were able to avoid a fork, but in the next example we encounter a situation in which we cannot.

List of all prefixes — *unavoidable forks*

Let function f, $f \in \mathrm{X} \to [\mathrm{X}]$, be defined by

$$\begin{array}{lcll} f.x & = & [x] & ,\ \mathsf{isempty}.x \\ & [] & x \vdash f.(\mathsf{ft}.x) & ,\neg\mathsf{isempty}.x. \end{array}$$

Then $f.x$ is the list of all prefixes of x (in decreasing order of length). Since f transforms a deque into a list (more precisely, a stack) of deques, the amortized costs of f are defined as follows:

$$\mathcal{A}[f](x) = \mathcal{T}[f](x) + \Phi^*.(f.x) - \Phi.x,$$

with $\Phi^*.[\,] = 0$ and $\Phi^*.(x \vdash xs) = \Phi.x + \Phi^*.xs$. Just as in the previous section there is a fork in the recursive alternative of f, which gives rise to the term $\Phi.x$ in the following recurrence relation:

$$\mathcal{A}[f](x) = \begin{cases} \mathcal{A}[\mathsf{isempty}](x) & ,\llbracket x\rrbracket = [\,] \\ \mathcal{A}[f](\mathsf{ft}.x) + \mathcal{A}[\mathsf{ft}](x) + \mathcal{A}[\mathsf{isempty}](x) + \Phi.x & ,\llbracket x\rrbracket \neq [\,]. \end{cases}$$

Since $\Phi.x$ is at most $\#\llbracket x\rrbracket$, it follows that $\mathcal{A}[f](x)$ is $O((\#\llbracket x\rrbracket)^2)$. Now, in contrast with the previous example, we cannot give an algorithmic refinement of f such that $\mathcal{A}[f]$ is linear; we even conjecture that it is impossible to refine f such that $\mathcal{A}[f]$ is linear[1]. This result seems somewhat disappointing when one realizes

[1] As a motivation for this conjecture, we briefly describe a failing attempt of ours: we assume that X and Φ are defined as in Hoogerwoord's refinement. To achieve linearity of $\mathcal{A}[f](x)$, we aim at linearity of $\mathcal{T}[f](x)$ and $\Phi^*.(f.x)$. This suffices because Φ is nonnegative. A simple way to establish linearity of $\Phi^*.(f.x)$ is to keep the Φ-values $O(1)$ for all elements of $f.x$. This means that the two components of all elements of $f.x$ should be roughly of equal length. Hence, $f.x$ should be a list of $\#\llbracket x\rrbracket{+}1$ "balanced pairs". From a study of this list, we conclude that it is impossible to generate it in linear time.

that $\mathcal{T}[f]$ itself is linear: with $\mathcal{A}'[f](x) = \mathcal{T}[f](x) - \Phi.x$, we have that $\mathcal{A}'[f](x)$ is $O(\#[\![x]\!])$, and, consequently, $\mathcal{T}[f](x)$ is $O(\#[\![x]\!])$ (using $\Phi.x \leq \#[\![x]\!]$). However, as we will argue in Section 5.6.3, it is not so bad that $\mathcal{A}[f]$ is quadratic.

Deque of all prefixes — *nested deques*

The following data refinement of f is also an instructive example:

$$\begin{array}{lcll} g.x & = & \mathsf{cons}.x.\mathsf{empty} & , \ \mathsf{isempty}.x \\ & [] & \mathsf{cons}.x.(g.(\mathsf{ft}.x)) & , \neg\mathsf{isempty}.x. \end{array}$$

Hence, $g.x$ is a deque of deques. In such a case we use the abstraction function in the definition of the amortized costs:

$$\mathcal{A}[g](x) = \mathcal{T}[g](x) + \Phi.(g.x) + \Phi^*.[\![g.x]\!] - \Phi.x.$$

Compared to f the additional amortized costs are linear and, therefore, we have that $\mathcal{A}[g](x)$ is also $O((\#[\![x]\!])^2)$, while $\mathcal{T}[g](x)$ is $O(\#[\![x]\!])$.

5.6.3 Comparison with doubly-linked list representation

In this section we compare the "pair of stacks" representation (Section 5.6.1) with the doubly-linked list representation (Section 4.5), which we call $S2$ and D, respectively.

First, some important differences:

(a) Algebras using $S2$ are efficient in the amortized sense only, whereas algebras using D are efficient in the worst-case sense.

(b) Algebras using D are destructive, whereas algebras using $S2$ are destructive only if the destructive implementation of stacks is used.

(c) Representation D uses about twice as much pointers as representation $S2$.

(d) Representation D supports an $O(1)$ implementation for $+\!\!+$, whereas representation $S2$ does not. Furthermore, selection of the n-th element of a list of length N takes $O(n)$ time for representation D, but it takes $O(N)$ amortized time for representation $S2$.[2]

Now, focusing on (a) and (c), we observe an interesting *trade-off* between time and space: representation $S2$ requires less space than representation D, but algebras using $S2$ are efficient in the amortized sense only whereas algebras using D are efficient in the worst-case sense. Thus, one can save space by

[2]Representation $S2$ supports a simple $O(1)$ implementation for rev. Representation D does not, but it can be easily extended to do so because of the symmetry in the l-links and r-links: by interchanging the role of these links the reverse is obtained; the present meaning of the links can be recorded by a boolean.

contenting oneself with amortized costs, which is as good as actual costs when one is interested in linear usage only.

In the previous section we have seen that the fact that $\mathsf{S2}$'s operations are efficient in the amortized sense only may lead to strange results for the amortized costs of programs with forks. For instance, the amortized costs of program f, which computes the list of all prefixes of its input list, are higher than its actual costs: $\mathcal{A}[f](x)$ is quadratic in $\#[\![x]\!]$ while $\mathcal{T}[f](x)$ is linear. The fact that $\mathcal{A}[f](x)$ is quadratic is however not so bad when we consider the cost of an implementation of f using doubly-linked lists. In that case, correct evaluation of f requires that the value of x is copied in each unfolding of $f.x$, which takes time quadratic in $\#[\![x]\!]$ (cf. Remark 4.2). Similarly, the amortized costs for the nonlinear program for $\mathsf{split}.x.n$ are higher than desired, viz. $O(\#[\![x]\!])$ instead of $O(n)$. But, again, when doubly-linked lists are used, execution of this nonlinear program requires that the value of x is copied one time to supply it once to take and once to drop. Hence, evaluation of $\mathsf{split}.x.n$ takes $O(\#[\![x]\!])$ time, which equals the bound for $\mathcal{A}[\mathsf{split}](x, n)$. So, the results for the amortized costs of these programs are not so strange after all.

Chapter 6

More on lists and trees

In many data structure designs, lists and trees play a vital role as interface between specifications in terms of sets or bags on the one hand and implementations in terms of arrays or pointers on the other. The use of lists and trees in these designs is often explained by means of pictorial descriptions, but—for our purposes—this approach has two major drawbacks.

A first problem is that correctness proofs of data structures based on pictures of lists and trees tend to be rather sloppy. We regard this as a consequence of the informal status of the pictures in such proofs, their meaning being vague or ambiguous. Moreover, pictorial descriptions are often incomplete because the pictures do not cover all cases. In this way, only an intuitive explanation of the implementation (in terms of arrays or pointers) is provided.

A more serious drawback is that the use of pictures does not support the calculational way of amortized analysis we have in mind. To be able to derive potential functions in a systematic way, we will use *patterns*—as formal counterpart of the use of pictures—to program operations of data structures. Using the appropriate patterns, the essential effect of a program on the *structure* of its arguments can be expressed formally and concisely. For operations on lists, operators like $\vdash$, $\dashv$, and $+\!\!+$ can be used to build patterns. For example, an operation may transform pattern $s +\!\!+ [a] +\!\!+ t$ into $[a] +\!\!+ s +\!\!+ t$.

To build patterns for trees, however, we do not have such an adequate set of operators. In formal definitions of operations on trees of type $\langle T\rangle$, for instance, a tree is either $\langle\,\rangle$ or of the form $\langle t, a, u\rangle$. This may be compared to a restricted view of lists in which a list is either $[\,]$ or of the form $a \vdash s$ (like lists in a functional language). To program more advanced operations on trees of type $\langle T\rangle$, we see that the counterparts of $\dashv$ and $+\!\!+$ are missing. Therefore, we will introduce alternative tree types such that, for instance, swapping the subtrees of node a can be expressed as the transformation of pattern $v \dashv \langle t, a, u\rangle$ into pattern $v \dashv \langle u, a, t\rangle$.

The goal of this chapter is to lay a foundation for the use of list and tree patterns in Part II by presenting (pointer) implementations for the relevant list and tree algebras.

6.1 Lists

In the list algebras considered thus far, an important class of operations has been ignored, viz. operations that modify a list around a specified element of the list. Typical examples are insertion after a specified element and deletion of an element, which may defined as follows for lists without duplicates:

$$\begin{array}{lcl} ins.b.a.(s +\!\!\!+ [a] +\!\!\!+ t) & = & s +\!\!\!+ [a,b] +\!\!\!+ t \\ del.a.(s +\!\!\!+ [a] +\!\!\!+ t) & = & s +\!\!\!+ t. \end{array}$$

A more primitive operation of this kind is *split*, where

$$split.a.(s +\!\!\!+ [a] +\!\!\!+ t) = (s,t),$$

in terms of which *ins* and *del* can be defined without the use of parameter patterns:

$$\begin{array}{lcl} ins.b.a.x & = & s +\!\!\!+ [a,b] +\!\!\!+ t \ \ [\![\ (s,t) = split.a.x \ ]\!] \\ del.a.x & = & s +\!\!\!+ t \ \ [\![\ (s,t) = split.a.x \ ]\!]. \end{array}$$

Note that the effect of these operations on the structure of their arguments is made explicit by the use of parameter patterns; for instance, the first definition of $del.a.s$ not only shows that a is removed from s but also that the order of the remaining elements is not altered.

Although the definitions of these operations employ patterns built from $[\cdot]$ and $+\!\!\!+$—instead of patterns built from $[\,]$ and $\vdash$, as is common for functional languages—we want to regard these definitions as (functional) programs. What is more, we will assume in Part II that these programs have constant time complexity. This "advanced" use of list patterns, however, requires some restrictions on the use of lists.

First of all, to be able to determine the position of an element a in list s in constant time, we will assume that

- list elements are of type $[0..N)$, for some fixed natural number N, and
- lists do not contain duplicates.

As will be shown in Section 6.1.1, these assumptions enable us to keep track of the position of elements of $[0..N)$ in the linked structures representing the lists (by means of an array of pointers with domain $[0..N)$).

Furthermore, in case an algebra provides operations for combining and splitting lists, such as $+\!\!\!+$ and *split*, we must guarantee that

- each element of $[0..N)$ occurs in at most one list.

This enables the use of *one* pointer array with domain $[0..N)$ for a collection of lists (see Section 6.1.2). Because of the usage of arrays, these implementations are all destructive, hence these list algebras may be used in a linear fashion only.

6.1.1 Stacks with deletion

For fixed N, $N \geq 0$, we consider the algebra of stacks with $\vdash$ replaced by $\dot{\vdash}$, and tl replaced by the above defined operation del:

$$(\,[\,[0..N)\,]\ \mid\ [\,], (=[\,]), \dot{\vdash}, hd, del\,).$$

Operation $\dot{\vdash}$ is a restricted version of $\vdash$: $a \dot{\vdash} s$ is defined (equal to $a \vdash s$) only for $a \notin s$. Likewise, $del.a.s$ is defined only if $a \in s$. Consequently, the lists in the range of this algebra are free of duplicates.

To obtain an $O(1)$ implementation of del, we represent a list of type $[\,[0..N)\,]$ by a pair consisting of a pointer to a doubly-linked list and an array. More precisely, the concrete type is

$$D\ \times\ ([0..N) \to D),$$

where $D = {}^\wedge\langle l{:}D, n{:}[0..N), r{:}D\rangle$. The coupling is then defined by $[\,] \simeq \langle \mathrm{nil}, f\rangle$ for all f, and (cf. Section 4.5):

$$\begin{array}{lll} s \simeq \langle p, f\rangle & \equiv & (\forall i : 0 \leq i < \#s : f[s.i]^\wedge.n = s.i) \\ & & \wedge\ \#s = (\mathrm{Min}\, i : 1 \leq i : p(^\wedge.r)^i{=}p) \\ & & \wedge\ s = list.p{\uparrow}\#s \\ & & \wedge\ (\forall i : 0 \leq i \leq \#s : p(^\wedge.r)^i = p(^\wedge.l)^{\#s-i}). \end{array}$$

for all nonempty s. That is, for $a \in s$, $f[a]$ points to the cell containing a, and both the l-links and the r-links form circular lists.

The following programs show how array f is used:

$$\begin{array}{lll}
\textsf{empty} & = & \langle \mathrm{nil}, ?\rangle \\
\textsf{isempty}.\langle p, f\rangle & = & p = \mathrm{nil} \\
\textsf{cons}.a.\langle p, f\rangle & = & |[\mathrm{var}\ h{:}D;\ \mathrm{new}(h) \\
 & & \ ;\ (\ h^\wedge := \langle h, a, h\rangle \qquad\qquad\quad ,p = \mathrm{nil} \\
 & & \quad [\!]\ h^\wedge := \langle p^\wedge.l, a, p^\wedge.r\rangle \\
 & & \qquad ;\ p^\wedge.l^\wedge.r, p^\wedge.r^\wedge.l := h, h \quad ,p \neq \mathrm{nil} \\
 & & \quad) \\
 & & \ ;\ \mathrm{return}(\langle h, f[a{:=}h]\rangle) \\
 & &]| \\
\textsf{hd}.\langle p, f\rangle & = & p^\wedge.n \\
\textsf{delete}.a.\langle p, f\rangle & = & |[\mathrm{var}\ h{:}D;\ h := f[a] \\
 & & \ ;\ h^\wedge.l^\wedge.r, h^\wedge.r^\wedge.l := h^\wedge.r, h^\wedge.l \\
 & & \ ;\ (\mathrm{skip}\ ,p \neq h\ \ [\!]\ \ p := p^\wedge.r\ ,p = h) \\
 & & \ ;\ (\mathrm{skip}\ ,p \neq h\ \ [\!]\ \ p := \mathrm{nil}\ ,p = h) \\
 & & \ ;\ \mathrm{dispose}(h);\ \mathrm{return}(\langle p, f\rangle) \\
 & &]|.
\end{array}$$

This implementation is clearly destructive. Note that $tl.s$ can be expressed as $del.(hd.s).s$ for nonempty s. Hence, tl is refined by tl, with $\mathsf{tl}.\langle p, f\rangle = \mathsf{delete}.(p^\wedge.n).\langle p, f\rangle$.

6.1.2 Partitions

The representation used in the previous section also allows $O(1)$ implementations of operations like $\dashv$ and lt, and even an operation like ins can be implemented in constant time (with $ins.b.a.s$ defined only if $a \in s$ and $b \notin s$). However, addition of operations that combine or split lists give rise to complications.

For example, suppose we want to use type $D \times ([0..N) \to D)$ as concrete type in a refinement of algebra

$$(\,[\,[0..N)\,] \mid [\,], (=[\,]), [\cdot], +\!\!+, hd, lt, del\,),$$

where use of $s +\!\!+ t$ is restricted to disjoint s and t so that the lists remain free of duplicates. To achieve $O(1)$ cost for $+\!\!+$, we have to combine two representations $\langle p, f\rangle$ and $\langle q, g\rangle$, say, in constant time. But combining f and g into one array cannot be done in constant time, unless f and g denote the same array! To support an efficient implementation of $+\!\!+$, we therefore use a *global* array f for a collection of mutually disjoint lists.

To describe implementations of algebras of this kind formally, we may consider algebras with $\Lbag\,[\,[0..N)\,]\,\Rbag$ instead of $[\,[0..N)\,]$ as data type. Accordingly, the operations on $[\,[0..N)\,]$ are then lifted to operations on $\Lbag\,[\,[0..N)\,]\,\Rbag$. For example, $s +\!\!+ t$ is lifted to the operation which transforms $B \oplus \Lbag s\Rbag \oplus \Lbag t\Rbag$ into $B \oplus \Lbag s +\!\!+ t\Rbag$, thus keeping the bag elements mutually disjoint. To represent such a bag of mutually disjoint lists (a "partition of $[0..N)$"), one array with domain $[0..N)$ can be used plus a pointer for each bag element.

In this thesis we will however not pursue such a formal approach to the implementation of partitions. As a final remark on this subject, we mention that the use of $[\cdot]$ must be restricted as well: creation of singleton $[a]$ is only allowed if a does not occur in any of the lists in use.

6.2 Binary trees

6.2.1 Top-down views

Formal definitions of binary trees are often limited to, what we will call, *top-down* views. An example is type $\langle T\rangle$, which has been introduced in Section 1.1.4. Trees of this type are either $\langle\,\rangle$ or of the form $\langle t, a, u\rangle$ with $t \in \langle T\rangle$, $a \in T$, and $u \in \langle T\rangle$. That is, type $\langle T\rangle$ is the smallest solution of

$$X: \quad X = \{\langle\,\rangle\} \;\cup\; X \times T \times X.$$

Another common top-down type of binary trees is the smallest solution of

$$X: \quad X = T \ \cup \ X \times X.$$

Trees of this type are of the form a or $\langle t, u \rangle$, hence the values reside in the leaves of the tree. (This tree type corresponds to the type of LISP objects, which are either atoms or pairs of LISP objects.) A top-down type with both internal and external nodes may be defined as the smallest solution of

$$X: \quad X = T_e \ \cup \ X \times T_i \times X,$$

where T_e corresponds to external nodes (leaves) and T_i to internal nodes. Of course there are many more top-down views around, but we will focus on type $\langle T \rangle$ in the sequel.

Although these top-down views suffice—in principle—to define any operation on binary trees, many operations cannot be described in a neat way. For instance, an operation that increments the value of the node that is last in the inorder traversal of a tree may be defined as follows in terms of type $\langle T \rangle$:

$$\begin{array}{lcll} inc.\langle t, a, \langle\,\rangle\rangle & = & \langle t, a{+}1, \langle\,\rangle\rangle & \\ inc.\langle t, a, u\rangle & = & \langle t, a, inc.u\rangle & , u \neq \langle\,\rangle. \end{array}$$

Formally, this definition is correct, but as starting-point for an efficiency analysis it has two drawbacks. One drawback is that it is not immediate from this definition that $inc.t$ leaves the structure of t intact. In general, the effect of an operation on the structure of its tree arguments is essential for its analysis. Moreover, suppose that we have a pointer representation for $\langle T \rangle$ in mind for which operation inc can be implemented by a simple $O(1)$ program. In that case, a recursive definition of inc is not a suitable starting-point for an efficiency analysis, because it does not look like an $O(1)$ program.

What we want is a description of inc that shows the relevant changes in the structure, just as the definition of list operation del shows that $del.a$ transforms pattern $s \mathbin{+\!\!+} [a] \mathbin{+\!\!+} t$ into pattern $s \mathbin{+\!\!+} t$. To enable such a description of inc, we will introduce a special tree type in the next section, where we will also present a pointer implementation of an algebra involving this tree type. Subsequently, we consider two tree types which are particularly suited for the definition of operations on subtrees of specified nodes.

6.2.2 A skewed view

Suppose we wish to perform a number of operations on binary trees that manipulate the rightmost path. To provide quick access to this path, we define type $\langle T \rangle_1$ as the smallest solution of

$$X: \quad X = [X \times T].$$

Just as we interpret elements of $\langle T \rangle$ as binary trees although they are just tuples, we interpret elements of $\langle T \rangle_1$ as binary trees although they are just lists:

$[\,]$ stands for the empty tree and $\langle t,a\rangle \vdash u$ for a nonempty tree with left subtree t, root a, and right subtree u. Formally, this type is "isomorphic" to type $\langle T\rangle$ because, on the one hand, we have as abstraction function from $\langle T\rangle_1$ to $\langle T\rangle$:

$$\begin{array}{lcl} [\![\,[\,]\,]\!] & = & \langle\,\rangle \\ [\![\langle t,a\rangle \vdash u]\!] & = & \langle [\![t]\!], a, [\![u]\!]\rangle, \end{array}$$

and, on the other hand, we have as representation function from $\langle T\rangle$ to $\langle T\rangle_1$:

$$\begin{array}{lcl} (\![\langle\,\rangle]\!) & = & [\,] \\ (\![\langle t,a,u\rangle]\!) & = & \langle (\![t]\!), a\rangle \vdash (\![u]\!). \end{array}$$

In addition, however, nonempty elements of $\langle T\rangle_1$ can be written in the form $u \dashv \langle t,a\rangle$, in which a is the last element of the inorder traversal of this tree. In terms of patterns of type $\langle T\rangle_1$, operation inc can thus be programmed as

$$inc.(u \dashv \langle t,a\rangle) = u \dashv \langle t, a{+}1\rangle.$$

From this program it is immediate that the structure of the tree remains the same. Moreover, it looks like a program with constant time complexity—which is indeed the case, as will be shown shortly.

Another nice property of type $\langle T\rangle_1$ is that its elements can be catenated: if $t \in \langle T\rangle_1$ and $u \in \langle T\rangle_1$, then also $t \mathbin{+\!\!+} u \in \langle T\rangle_1$. This property can, for instance, be used to generate the inorder traversal in a simple way:

$$\begin{array}{lcl} io.[\,] & = & [\,] \\ io.(u \dashv \langle t,a\rangle) & = & io.(u \mathbin{+\!\!+} t) \dashv a, \end{array}$$

using that $\overline{[\![t \mathbin{+\!\!+} u]\!]} = \overline{[\![t]\!]} \mathbin{+\!\!+} \overline{[\![u]\!]}$. This program is as good as tail recursive—in contrast with the definition of $\overline{\cdot}$ for type $\langle T\rangle$, which is not even linearly recursive.

To support the above-mentioned operations we implement algebra

$$\mathsf{T1} = (\,\langle T\rangle_1 \mid [\,], (=[\,]), [\cdot], \mathbin{+\!\!+}, hd, tl, lt, ft\,).$$

Since $\langle T\rangle_1 = [\langle T\rangle_1 \times T]$, this algebra may be considered as an instance of algebra CDQ (see Section 4.5):

$$\mathsf{CDQ} = (\,[U] \mid [\,], (=[\,]), [\cdot], \mathbin{+\!\!+}, hd, tl, lt, ft\,).$$

However, instantiation of the refinement of CDQ with $U := \langle T\rangle_1 \times T$ does not give a concrete refinement of $\mathsf{T1}$ because type $\langle T\rangle_1$ is not concrete. To get around this problem, we have to instantiate the refinement of CDQ with $U := D \times T$, where D is the concrete type refining $[U]$. After renaming we thus obtain the following pointer type:

$$R = {}^\wedge\langle l{:}R, a{:}R{\times}T, r{:}R\rangle.$$

The programs for the operations of $\mathsf{T1}$ can now be obtained from the programs for CDQ. This yields a destructive implementation of $\mathsf{T1}$, for which all operations have constant time complexity. The price to be paid is an extra pointer per node, viz. three pointers instead of two for the standard implementation of $\langle T\rangle$ as described in Section 4.3.

In Chapter 9 we will use algebra $\mathsf{T1}$ to program bottom-up skew heaps.

6.2.3 Root-path views

Another example of an operation that is often supported in constant time is removal of a specified subtree. Assuming that a occurs exactly once, removal of the subtree with root a may be defined as follows in terms of type $\langle T\rangle$ (using $\in$ to determine the location of a):

$$\begin{array}{lcll} rem.a.\langle t,b,u\rangle & = & \langle\,\rangle & ,a=b \\ & [] & \langle rem.a.t,b,u\rangle & ,a\in t \\ & [] & \langle t,b,rem.a.u\rangle & ,a\in u. \end{array}$$

As before with the definition of inc, this definition of rem in terms of $\langle T\rangle$ is not what we want when rem corresponds to a simple $O(1)$ program at pointer level.

We obtain a more appropriate definition for rem by defining it in terms of type $\langle T\rangle_2$, which is the smallest solution of

$$X:\quad X=[X\times T\cup T\times X].$$

Although elements of this type are actually lists, they can be interpreted as binary trees according to the following abstraction function:

$$\begin{array}{lcl} [\![\,[\,]\,]\!] & = & \langle\,\rangle \\ [\![\langle t,a\rangle\vdash u]\!] & = & \langle [\![t]\!],a,[\![u]\!]\rangle \\ [\![\langle a,u\rangle\vdash t]\!] & = & \langle [\![t]\!],a,[\![u]\!]\rangle. \end{array}$$

Since $[\![\langle t,a\rangle\vdash u]\!] = [\![\langle a,u\rangle\vdash t]\!]$, the same tree in $\langle T\rangle$ is represented by many different "trees" in $\langle T\rangle_2$. Consequently, a function definition in terms of type $\langle T\rangle_2$ does not necessarily induce a function on $\langle T\rangle$. A sufficient condition to ensure that a function f on $\langle T\rangle_2$ induces a function on $\langle T\rangle$ is that $[\![f.t]\!]=[\![f.u]\!]$ should hold whenever $[\![t]\!]=[\![u]\!]$.

In terms of type $\langle T\rangle_2$, removal of the subtree rooted in a can now be described as

$$rem.a.(v +\!\!+ \langle t,a\rangle\vdash u)=v.$$

A similar, but more primitive operation is

$$split.a.(v +\!\!+ \langle t,a\rangle\vdash u)=(t,v,u).$$

In these definitions v represents more than a tree: it also records the location where the subtree with root a resided prior to removal. Or, formally: if $split.a.x=(t,v,u)$ then $[\![x]\!]=[\![v +\!\!+ \langle t,a\rangle\vdash u]\!]$, which means that the tree represented by x can be reconstructed from a and $split.a.x$.

As counterpart of rem we have "catenation" of trees: for t and u in $\langle T\rangle_2$, also $t +\!\!+ u$ denotes a tree in $\langle T\rangle_2$. In terms of type $\langle T\rangle$, catenation of t and u means that an empty subtree of a node in t is replaced by u; which one is replaced is determined by t. For example, catenation of $[\langle t_0,a_0\rangle,\langle a_1,t_1\rangle,\langle a_2,t_2\rangle]$

and u corresponds to replacing the left subtree of node a_2—which is empty—by u.

Note that rem is not defined for all elements of $\langle T\rangle_2$; $rem.a.(v +\!\!+ \langle a,u\rangle \vdash t)$, for example, is not defined. This is no problem as long as a function value is defined for at least one representation of each tree in $\langle T\rangle$. For a particular function we choose the elements of $\langle T\rangle_2$ that are best suited to define it.

To define operations that manipulate specified subtrees, often a type like

$$\langle T\rangle_3 = [\langle T\rangle \times T \cup T \times \langle T\rangle] \times \langle T\rangle$$

suffices instead of the more intricate type $\langle T\rangle_2$. An element of this type represents a tree according to the following abstraction:

$$\begin{array}{lcl} [\![([\,],x)]\!] & = & x \\ [\![(\langle t,a\rangle \vdash u,x)]\!] & = & \langle t,a,[\![(u,x)]\!]\rangle \\ [\![(\langle a,u\rangle \vdash t,x)]\!] & = & \langle [\![(t,x)]\!],a,u\rangle. \end{array}$$

In terms of type $\langle T\rangle_3$, a tree x in which a occurs can be decomposed in exactly one way as $x = (v, \langle t,a,u\rangle)$. That is, this representation allows us to dissect tree x in the subtree rooted in a with left subtree t and right subtree u, and the remainder of x, called v, which encodes the *root-path* of a in x. This contrasts with type $\langle T\rangle_2$, for which these components are not uniquely determined. As programs for rem and $split$ we now have

$$\begin{array}{lcl} rem.a.(v,\langle t,a,u\rangle) & = & (v,\langle\,\rangle) \\ split.a.(v,\langle t,a,u\rangle) & = & (([\,],t),(v,\langle\,\rangle),([\,],u)). \end{array}$$

In Part II we will often abbreviate $([\,],x)$ to x, and $(v,\langle\,\rangle)$ to v.

Pointer representations

We briefly discuss a pointer representation of $\langle T\rangle$ that supports the kind of decomposition required for patterns of type $\langle T\rangle_2$ and $\langle T\rangle_3$. The pointer type is

$$P = {}^\wedge\langle l{:}P, a{:}T, u{:}P, r{:}P\rangle.$$

The coupling $\simeq$ between $\langle T\rangle$ and P is defined as follows

$$x \simeq p \;\equiv\; x = tree.p \;\wedge\; DL.p \;\wedge\; p^\wedge.u = \mathrm{nil},$$

where

$$\begin{array}{lcl} tree.\mathrm{nil} & = & \langle\,\rangle \\ tree.p & = & \langle tree.(p^\wedge.l), p^\wedge.a, tree.(p^\wedge.r)\rangle \quad ,p \neq \mathrm{nil}, \end{array}$$

and $DL.p$ expresses that p points to a "doubly-linked tree":

$$\begin{array}{lcl} DL.\mathrm{nil} & \equiv & \mathrm{true} \\ DL.p & \equiv & (p^\wedge.l \neq \mathrm{nil} \;\Rightarrow\; p^\wedge.l^\wedge.u = p) \;\wedge\; DL.(p^\wedge.l) \\ & & \;\wedge\; (p^\wedge.r \neq \mathrm{nil} \;\Rightarrow\; p^\wedge.r^\wedge.u = p) \;\wedge\; DL.(p^\wedge.r). \end{array}$$

As remarked before, an element v of $[\langle T\rangle \times T \cup T \times \langle T\rangle]$ represents more than a tree in $\langle T\rangle$. To represent v we use in addition to a pointer p to the root of v, a pointer q pointing to the last node of v (if $v = [\,]$, then $q = \text{nil}$). In this way, four pointers are used to represent triple (t, v, u), which is the result of $split.a.x$: two for v, one for t, and one for u. To obtain these pointers in constant time, we assume that type T is of the form $[0..N)$, and that trees are free of duplicates. As in Section 6.1, this enables us to use a global array f of type $[0..N) \to P$ such that, given a, the pointers to the roots of t and u are $f[a]^\wedge.l$ and $f[a]^\wedge.r$, respectively, and $f[a]^\wedge.u$ points to the last node of v. Hence, given a and a pointer p to the representation of x, the representation of $split.a.x$ can be obtained from array f in constant time.

6.3 Trees and forests

Instead of binary trees in which each node has two subtrees, many data structure designs employ a view of trees in which each node has an unbounded (yet finite) number of subtrees. In the literature, a collection of trees is called a *forest*; in this terminology, a *tree* consists of a root node and a forest of subtrees. In the next section, we present some common types of trees and forests. Subsequently, we generalize the root-path views of binary trees defined in Section 6.2.3 to root-path views of trees and forests.

6.3.1 Top-down views

There are many top-down views of trees and forests. An important distinction between these views is whether the order of the subtrees is taken into account or not. In formal definitions of tree types, this corresponds to either a list or a bag/set of subtrees. Two such tree types are $\langle T\rangle_4$ and $\langle T\rangle_5$, defined as smallest solutions of equations

$$X: \quad X = T \times [X],$$

and

$$X: \quad X = T \times \lbag X \rbag,$$

respectively. An element $\langle a, t\rangle$ of $\langle T\rangle_4$, with $a \in T$ and $t \in [\langle T\rangle_4]$, stands for a nonempty tree with root a and the trees in *list* t as subtrees. Similarly, an element $\langle a, t\rangle$ of $\langle T\rangle_5$ denotes a nonempty tree with root a and the trees in *bag* t as subtrees. Elements of type $[\langle T\rangle_4]$ and $\lbag \langle T\rangle_5 \rbag$ are forests.

In many data structures, trees are required to be free of duplicates, and also trees in a forest are required to be disjoint. Instead of type $\langle T\rangle_5$, a type like $\langle T\rangle_6$ may then be used, which is defined as the smallest solution of

$$X: \quad X = T \times \{X\}.$$

The advantage over type $\langle T\rangle_5$ is that standardized set notations can be used to define operations on trees of type $\langle T\rangle_6$.

6.3.2 Root-path views

In a similar way to that in Section 6.2.3, we now introduce root-path views of types $\langle T\rangle_4$ and $\langle T\rangle_6$ to facilitate the definition of operations that modify specified subtrees.

We first consider the simpler type $\langle T\rangle_6$. To program the removal of a specified subtree for this type, it suffices to introduce type

$$\langle T\rangle_7 = [\langle T\rangle_6].$$

A *nonempty* list of this type represents a tree in $\langle T\rangle_6$ according to the following abstraction:

$$\begin{array}{lcll} [\![[x]]\!] & = & x & \\ [\![\langle a,t\rangle \vdash v]\!] & = & \langle a, t \cup \{[\![v]\!]\}\rangle & , v \neq [\,]. \end{array}$$

(The empty list (of type $\langle T\rangle_7$) may be viewed as an empty tree.) In terms of this type, a tree in which a occurs exactly once can be written in the form $v \dashv \langle a,t\rangle$ in precisely one way. The program for removal is thus:

$$rem.a.(v \dashv \langle a,t\rangle) = v,$$

in which v corresponds to the root-path of a.

Next we consider type $\langle T\rangle_4$. In this case, a more intricate type is required to define *rem* because the order of the subtrees must be taken into account. The following type is appropriate:

$$\langle T\rangle_8 = [\,[\langle T\rangle_4] \times T \times [\langle T\rangle_4]\,] \times \langle T\rangle_4.$$

The abstraction function is

$$\begin{array}{lcl} [\![([\,],x)]\!] & = & x \\ [\![(\langle t,a,u\rangle \vdash v, x)]\!] & = & \langle a, t +\!\!+ [[\![(v,x)]\!]] +\!\!+ u\rangle. \end{array}$$

Hence, (v,x) represents the $(\#t+1)$-st subtree of a, located between those subtrees in t and those in u. In terms of type $\langle T\rangle_8$, a tree in which a occurs exactly once can be written in the form $(v,\langle a,t\rangle)$ in precisely one way, which leads to the following program for removal:

$$rem.a.(v,\langle a,t\rangle) = (v,\langle\,\rangle).$$

Again, v encodes the root-path of a.

Pointer representations

Pointer representations for types like $\langle T\rangle_4$ and $\langle T\rangle_6$ are in essence much the same as those for binary tree types. An appropriate pointer type for $\langle T\rangle_4$ that also supports root-path view $\langle T\rangle_8$ is, for instance (cf. type P in Section 6.2.3):

$$Q = {}^\wedge\langle a{:}T, u{:}Q, s{:}[Q]\rangle.$$

For a pointer p of this type, $p^\wedge.u$ provides access to the "parent" node and $p^\wedge.s$ represents the forest of subtrees of node a. To support efficient deletion of arbitrary subtrees, a doubly-linked list representation can be used for type $[Q]$. This amounts to four pointers per node.

In case $T = [0..N)$ and access to subtrees must be supported in constant time, an array of type $[0..N) \rightarrow Q$ is required in addition (see Section 6.2.3). Then a total of five pointers per node is used.

6.4 Overview of tree types

Recall that $(\mu x : P : E)$ denotes the smallest solution of $x : \; P \wedge x = E$. Throughout this chapter we have defined the following types of binary trees:

$$\begin{array}{lcl}
\langle T\rangle & = & (\mu X : \langle\,\rangle \in X : X \times T \times X) \\
\langle T\rangle_1 & = & (\mu X :: [X \times T]) \\
\langle T\rangle_2 & = & (\mu X :: [X \times T \cup T \times X]) \\
\langle T\rangle_3 & = & [\langle T\rangle \times T \cup T \times \langle T\rangle] \times \langle T\rangle.
\end{array}$$

Each binary tree of type $\langle T\rangle$ is represented by exactly one binary tree in $\langle T\rangle_1$ but by many different "binary trees" in $\langle T\rangle_2$ and $\langle T\rangle_3$.

Furthermore, we have defined the following types of nonempty trees:

$$\begin{array}{lcl}
\langle T\rangle_4 & = & (\mu X :: T \times [X]) \\
\langle T\rangle_5 & = & (\mu X :: T \times \Lbag X \Rbag) \\
\langle T\rangle_6 & = & (\mu X :: T \times \{X\}) \\
\langle T\rangle_7 & = & [\langle T\rangle_6] \\
\langle T\rangle_8 & = & [\,[\langle T\rangle_4] \times T \times [\langle T\rangle_4]\,] \times \langle T\rangle_4,
\end{array}$$

where $\langle T\rangle_4$ and $\langle T\rangle_8$ as well as $\langle T\rangle_6$ and $\langle T\rangle_7 \setminus \{[\,]\}$ are isomorphic.

In Part II we will use these tree types without bothering about their pointer implementations. We claim, however, that all basic tree operations used in Part II can be supported in constant time, using the techniques of this chapter.